Aspect oriented service composition for telecommunication applications

Niemöller, Jörg

Publication date:
2016

Document Version
Publisher's PDF, also known as Version of record

Aspect Oriented Service Composition for Telecommunication Applications

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan Tilburg University op gezag van de rector magnificus, prof. dr. E. H. L. Aarts, in het openbaar te verdedigen ten overstaan van een door het college voor promoties aangewezen commissie in de aula van de Universiteit op vrijdag 8 april 2016 om 10.15 uur

door Jörg Niemöller

geboren op 16 oktober 1970 te Dortmund, Duitsland.

Promotores:
prof. dr. ir. M. P. Papazoglou
prof. dr. W. J. A. M. van den Heuvel

Overige Leden van de Promotiecommissie:
prof. dr. G. Ortiz Bellot
prof. dr. P. Giorgini
prof. dr. ir. M. J. van Sinderen

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Graduate School for Information and Knowledge Systems (Dissertation Series No. 467), and CentER, the Graduate School of the Faculty of Economics and Business Administration of Tilburg University.

ISBN: 978 90 5668 468 6

Preface

The work on service composition at Ericsson Research in Herzogenrath, Germany, is the origin of this dissertation. The author was part of the research team, which contributed considerable advances in the study of dynamic service composition and the use of SOA methodology and principles in telecommunication service environments. The work resulted in the Ericsson Composition Engine prototype and later in a product of the same name. It is still actively developed by Ericsson at the time of finishing this dissertation. Based on its concepts, various challenges in dynamic and heterogeneous service environments were studied. The original research focus was naturally on telecommunication, but the results became useful in a broader context.

Jörg Niemöller, has contributed to this research on service composition for the telecommunication service layer. One particularly challenging topic was the dynamic management and control of service level agreements. In particular, technical solutions for monitoring required metrics of composite service applications were investigated. The composite nature of the application and the data-driven, dynamic service composition mechanism of the Ericsson Composition Engine was the initial challenge and at the same time part of the solution. It did allow the implementation of a layer for reflection and automated modification. The proposed solution was basically an implementation of Aspect Oriented Programming (AOP). It was a variant of AOP, which allows to reason about the application instance at run-time and apply changes dynamically. This variant is referred to as online weaving. It was rarely used in practical deployments of AOP, and thus, its characteristics and potential applications were not fully understood. This dissertation is therefore filling some gaps and it answers in particular questions about the practical feasibility of AOP with dynamic online weaving. A complete AOP based development environment with an online weaving enabled composition execution engine was developed. Performance considerations and optimizations receive special attention, because real-time behavior of services is a particularly important requirement in telecommunication.

This dissertation summarizes the results of multiple years of research. It extends the composition paradigm with techniques for automated real time reflexion and instantaneous run time modification of applications. Although initially investigated with telecommunication services in mind, the results are highly relevant also for emerging domains, such as pervasive computing and the Internet of things. The majority of the concept development and implementation was performed between 2009 and 2011 at Ericsson Eurolab Germany in Herzogenrath. The validation of the concept and the work on the dissertation

was finished at Ericsson Research in Stockholm, Sweden. Writing a dissertation about this research at Ericsson did require to go considerably further than the internal projects would have required. For example, the newly developed concepts needed to be anchored within a broad base of exiting research. Furthermore, a particularly thorough investigation of performance was executed.

I would like to thank Ioannis Fikouras for the initial contact to Professor Mike Papazoglou and the University of Tilburg and for being a good friend and support also after the composition work was finished. I would like to express special thanks also to Roman Levenshteyn, the core architect and developer of the Ericsson Composition Engine research prototype, and the members of the service composition research team: Konstantinos Vandikas, Raphaël Quinet, Eugen Freiter and Friedhelm Ramme. They were always available for discussions and provided support, when needed. And last but not least, I would like to express my special thanks to Muzzamil Aziz Chaudhary and Meshkatul Anwer, who helped completing the proof of concept implementation of solutions developed within this dissertation. Their work did help creating a rich environment for aspect oriented application development and it did facilitate the extensive performance validation and the demonstration of practical use cases.

Contents

Chapter 1

Introduction

"Communication is a basic human need". This statement is credited to Lars Magnus Ericsson [1]. It originates in the 19th century, when telegraphy was state of the art in communication technologies and the telephone was just invented. It is today as relevant as ever since. It refers to communication being in the center of our social interactions and crucial for mastering our day to day life. This is particularly true in times when an enormous amount of information is available to each of us everywhere and anytime.

Within only a few years the Internet has become a powerful tool that has changed the way we do business and the way we communicate. It opens an international and global dimension to communication and it provides access to data and services. In particular, it has become a universal source of information and media, available on demand from any place. The only prerequisite is the availability of network access, and today, except for exceptionally remote locations, some sort of access is available practically everywhere.

This is a revolution repeating itself. The time from key inventions to broad usage was longer, but nevertheless telecommunication originally had a similar disruptive impact on the way individuals communicate and interact with each other. The world became the often referred to global village due to instantaneous availability of communication entirely independent of the physical distance. And still today, the solutions and services based on telecommunication technologies allow users to intuitively access the Internet, contact other persons and reach out to information, devices and services from anywhere.

People and businesses around the globe have embraced the broad availability of services and information. Thus, telecommunication solutions are a commodity and basic enabler of business and life in networked societies. This created large and highly competitive markets for service providers and system vendors. Consequently, it generates considerable pressure on technological development and innovation cycles in order to satisfy the need for information and communication, and enabling users accessing and mastering it. A great number of technologies have been developed in order to address these challenges. Today these technologies are broadly deployed while the need for further improvements and innovation is continuously high. For instance, the capabilities of mobile access technologies have dramatically changed within the past 15 years. This did enable new service experiences, aided by new classes of personal devices.

There are three major industries that have formed around these communication and information challenges: Telecommunication, IT and Enterprise. Historically, the telecommunication industry is focused on efficient, available and affordable communication between two individuals. The great achievement here is the availability of services and network access for everybody. The Enterprise domain deals mainly with managing and analyzing information, as used in a business-to-business context. Control of a business and methods to operate it efficiently are the dominating challenges of this domain. The IT domain focuses on data storage, data access and data manipulation.

All three domains have in common that they deal with information, its efficient distribution and services around it. Nevertheless, they focus on different aspects, which did lead to unique business models and requirements. Starting at different historical roots they have in parallel developed solutions based on their unique domain specific challenges. Today, the technical solutions applied in the three industry domains converge in the sense that they are increasingly based on the same technological foundations. Consequently, new technological developments, such as virtualization of key network assets, impact all domains alike. As a result, the respective industries are not clearly distinguished any more from a technical perspective. Business models follow this trend and break out of traditional vertical market domains. Offered solutions spread horizontally and create a broad Information and Communication Technology (ICT) environment.

1.1 Business Needs of Digital Service Providers

Business in the telecommunication, IT and enterprise domain is to a great extent centered around the notion of a service. Services are the economic good that is sold by service providers and bought by their customers. A service can, for example, be the processing of data for answering a user's question or it can be the connection of two users, or rather their devices, which ultimately allows them to communicate directly with each other. A unique set of service offerings defines a business that is serving users in a telecommunication, IT or enterprise context. Furthermore, innovation is frequently embodied either directly in new service ideas or indirectly in technical advances that then enable new types of service or improve services by giving them unique new characteristics. Either way a service providing company will need to create attractive offers that addresses changing market needs and that distinguishes it from its competition. It is in this respect seldom a completely new service offer that immediately has a vast impact on the market. The vast majority of innovation consists of small improvements by which a provider stays competitive.

Telecommunication services today are to a great extent a commodity. Typical services, such as voice calls and short messaging, are based on a well standardized infrastructure that was developed for decades reaching a very mature level of high quality, high availability and the capability to provide high volumes at low costs. These services are available practically everywhere and for everybody at an affordable price and at decent constant quality. But the fact that they are offered by every service provider in a similarly satisfactory and mature way does not allow for a great degree of competitive distinction. Since a while

also mobile data access to the public Internet and related basic services, such as mobile email, belong into this category of being a commodity. In general, it can be observed that every service with a high market penetration and user demand sooner or later becomes a commodity. For service providers with the ambition to improve and extent their business this situation translates into a constantly high innovation pressure.

In additions to the challenges of a commodity offering there are still services that are major distinguishing factors. Often these services provide access to unique content while the telecommunication network is used as a reliable infrastructure for service access and content delivery. Video and audio services accessed from the mobile device are a typical example. Access to popular social networking platforms on the public Internet is another one. While in the past these scenarios were entirely or to great extent served by a single company, for example the network operator, the resulting value chain now consists of contributions from various companies. The content is for example owned and provided by one specialized company while the access and delivery is provided by the company that owns a suitable network. Also the direct relationship to the end user is now often maintained by each involved company separately. This value chain is realized by a stack of services or it can be seen as a process. Each layer in the stack or each constituent service in the process might be operated by a different company in a technical and business sense. New popular services are often enabled by the network operator through improved capabilities of the network, but the actual content the user is interested in, is delivered over the top by another company.

Every company in this the value chain can improve its business in two dimensions: First of it needs to defend its position in the value chain and gain as much market share for its role. The over the top service provider needs to stay more attractive to users than similar offers. The network operator needs to defend its role of access provider and carrier against competing operators. This means constant improvement of the core services is needed to keep up an attractive offers. The second alternative would be to increase the share a company has of the value chain. This would effectively mean to extend the role and compete on different layers of the services stack or different places in the process.

For service providers in general, and therefore also for telecommunication operators, optimizing the return of investment from the offered services is a problem with multiple dimensions. For commodity service the cost needs to stay low, because there is not much room for growth in a saturated market. The capital expenditure (CAPEX) stays low when investing in an efficient infrastructure that can deliver high volumes of services at low cost per service usage. The operational expenditure (OPEX) stays low if the service environment needs low manual intervention. Therefore technologies that provide efficient design, deployment and maintenance will directly contribute to the service provider's business result.

For a good return of investment from service innovation it is primarily important to have a short time to market. Being early on the market with a new service means a competitive, and thus, income edge until also the competition has comparable offers and this service also becomes a commodity. This means that it is highly important that the time from identifying the business opportunity and the initial idea for a service offer until

it is available to customers needs to be as short and at the same time as cost efficient as possible.

Long tail services [2] are a special example, where a high degree of cost efficiency is needed. They are unique and highly customized services. Each of them targeting only a small number of potential users. These services are usually highly specific to the needs of this target user group. Often they are generated as a customized version of a more generic service. Due to their low volume and high specialization, these services will usually never become a commodity. Nevertheless, because of the small number of usages, the costs for developing them needs to be exceptionally small in order to generate a financial gain. Business in a long tail market often works by offering a vast amount of these specialized services at low volume and low average income from each, but also with low effort to produce and deliver the service. A good business result can still be generated by adding up the high number of small incomes.

1.2 The Impact of Service Composition and SOA

Service composition techniques can address the business challenges of service providers as mentioned above. They are part of a big group of technologies that facilitate component based software development (CBSD) [3] within a service oriented architecture (SOA) [4, 5]. These principles and technologies allow, among other properties, a high degree of modularization in service development with a clear separation of concerns combined with a high degree of abstraction. All these properties matter for achieving business gains through efficiency in building and maintaining a service offering.

Service composition in particular facilitates distribution of development efforts. This refers to constituent services within a composition being developed independently and existing components being used and re-used in many different service application contexts. The role of service composition in the development process is the final assembly of service applications from these constituent services. The term "service application" is used particularly in order to underline that a software application is build, that embodies and implements a service offer.

Service composition technologies usually operate on a high abstraction level, while implementation details are encapsulated in the constituent service modules. This contributes to an easier understanding of the application as a whole and to better control of its complexity. Because of these properties, it can significantly reduce the time needed for implementation while reaching higher software quality. The result of this modular and abstract development process therefore often means faster time to market at lower overall costs throughout the service life-cycle.

A consequence of this highly modular development infrastructure can be a specialization of service providers by taking the role of a vendor of service components. They specialize on particular features while others focus on assembling a service application. As a result, the aforementioned value chains can become even more partitioned and complex. In fact, highly elaborate value chains and business partnerships are possible within the context of

a single service application that is finally offered to the end-user. The technological details of a SOA based service infrastructure is what ultimately enables control over the complex development processes and diverse relationships to constituent service suppliers. Therefore, it is a central enabler for reaching cost advantages by outsourcing tasks to specialists.

Popular and widely used examples of service composition technologies are the Business Process Execution Language (BPEL) [6] and the Business Process Modeling Notation (BPMN) [7]. Together with their respective execution engines and development tools, these are the technical foundation for most of today's commercial deployments of service composition. They are an integral part of a service oriented architecture.

BPEL and BPMN define languages that allow to specify service compositions and they also define the respective execution semantics. Their expressiveness used to define and just communicate business processes as the topmost outline of the service application. Especially the first version of BPMN was mainly used this way due to missing uniquely standardized execution semantics. They were added in version 2.0 (BPMN) [7]. The related development process of a service application is done from top down. This means that first the idea for a service application is roughly expressed as a high level business process. Then, further details is successively added and the high level constituent components used in this process are implemented using a suitable modularization and distribution of concerns. This is a highly innovation-friendly process providing consistent tools for defining a service application throughout it's entire life-cycle and most layers of abstraction.

1.3 Telecommunication Service Composition

A unique service composition technology was developed specifically for service applications residing in a service layer of telecommunication networks. This technology is embodied in the Ericsson Composition Engine (ECE) [8, 9, 10, 11]. Its core is a composition language that is conceptually a general purpose language, but it also natively supports unique concepts of the telecommunication network and its unique service infrastructure. This includes, for example, the unique role of services within end-to-end communication contexts that are typical in telecommunication networks. The base of this service composition technology is the introduction of a service model for telecommunication services that follows SOA principles. Consequently, it allows telecommunication services providers to gain the business advantages of deploying a service oriented architecture. Applications can be developed following a composition methodology using extensive modular design and abstraction.

In the scope of this work, the term Ericsson Composition Engine (ECE) refers to the prototype implementation developed at Ericsson Research rather than the Ericsson product of the same name. The product called Ericsson Composition Engine is related, but it has a much wider scope and allows various development models for service applications. A composition based approach as defined by the research version of the Ericsson Composition Engine is just one out of many capabilities.

An important feature of the Ericsson Composition Engine is the support of a great

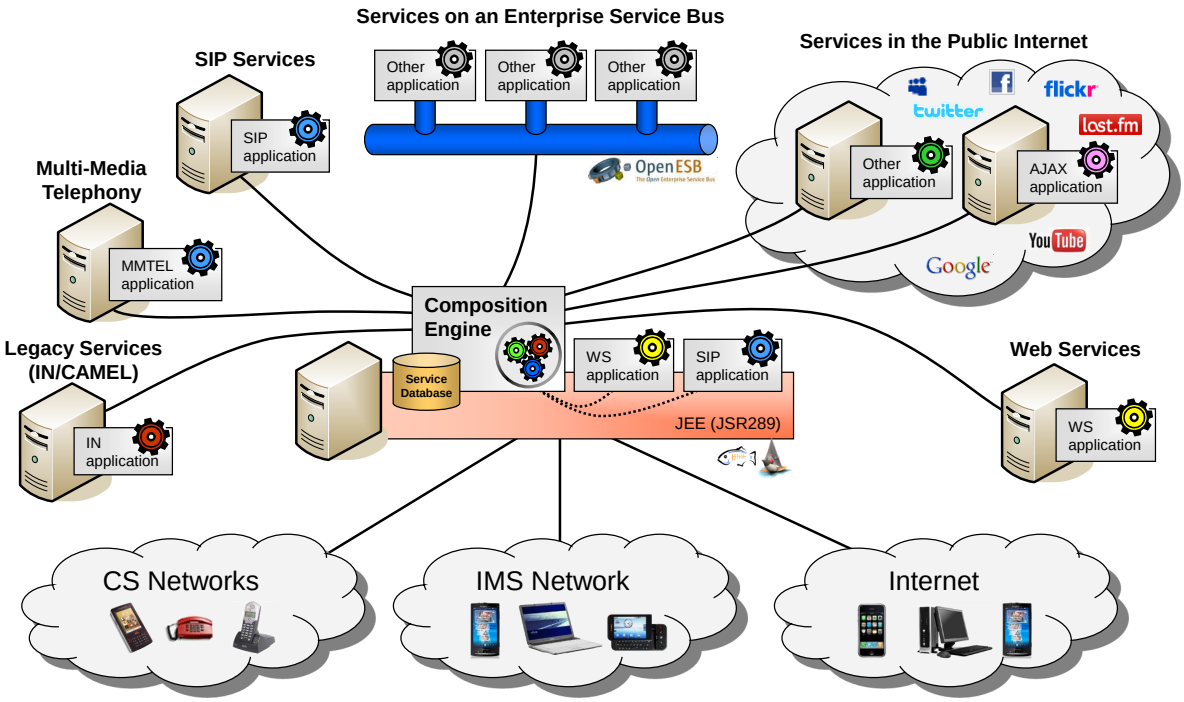

Figure 1.1: Overview of the Ericsson Composition Engine within the supported service technologies and potential accesses

number of different service technologies as shown in Figure 1.1. Composite applications can be composed from constituent service that are based on any of these technologies. A composite application can itself also be published, and therefore, it can be used as a service based on any of these technologies. Figure 1.1 also shows the different networks from which the composite application itself can be invoked and used. This enables the Ericsson Composition Engine to natively operate in heterogeneous service environments. This ability practically allows that a great number of available new and legacy components that can be composed into new applications. It also means a high flexibility of the resulting composite application with respect to usage contexts. Consequently, a great amount of already available and often highly customized legacy applications preserving proven business logic and extending investment lifetimes.

The capability of the Ericsson Composition Engine to natively support various service technologies is a different way of thinking compared to the web-service only service environments in which BPEL and BPMN based compositions used to operate. Their usual solution for to legacy support is using web service wrappers. This approach would not be possible for many telecommunication service technologies due to differences in the roles of services, their invocation and their usage topologies. These differences were the main reason why Ericsson has developed a telecommunication specific composition technology. Nevertheless, the Ericsson Composition Engine enables to use of composition techniques and SOA methodology also for typical telecommunication service applications.

1.4 The Promise of Aspect Orientation

The possibilities to modularize the development process and to control complex value chains are the main reasons why service composition and a SOA based service infrastructure are great assets to the business of service providers in telecommunication, IT and enterprise domains. It directly contributes to business results and allows to create a competitive service offering. However, there are challenges left, that SOA and Service Composition alone cannot address.

A good modularization is reached through an efficient separation of implementation concerns. Unfortunately, not all implementation concerns can be modularized in a straightforward way. The method usually applied is functional decomposition The various functions and sub-functions of the application are separated into modules. However, some frequent functional concerns cannot be separated from each other. Furthermore, there are also non-functional concerns that have not clear allocation in the functional modules. The necessary result of a purely functional decomposition is that the implementation of these concerns will be scattered across the application and modules become tangled with other modules. This property of an implementation and the concerns that causes it is refereed to as "cross-cutting" [12]. It leads to complex inter-dependencies and poorly modularized implementations despite the use of SOA and composition methodology. Important business concerns such as, charging and billing for service usage, logging of user actions or automated control of service metrics, are frequently cross-cutting.

Using service composition alone cannot reduce the effects of cross-cutting. It does not provide suitable tools to approach it in a systematic way. This is fundamentally critical, because composition environments depend on the availability of constituent services modules that are composable in different application contexts. This is the case if the constituent service is kept simple with as less mutual dependencies as possible. Cross-cutting concerns avoid that such constituent services can be implemented. Service modules become rather complex with many inter-dependencies. This means cross-cutting concerns effectively limit composability. Therefore, the benefits of using service composition can be significantly reduced. As a result, complexity caused by a need for detailed technical coordination is still high resulting in time-consuming, error-prone and ultimately costly development.

Aspect Oriented Programming (AOP) was proposed as an answer to this problem. It is a programming technique that allows to also separate cross-cutting concerns in a structured way and to reach a modular implementation nevertheless. Functional decomposition is not the only modularization technique for a service application any more, but AOP enables further complimentary ways to reach decomposition. This is the ultimate promise of Aspect Oriented Programming. It allows to keep the benefits of SOA and service composition even when facing cross-cutting concerns.

Aspect oriented methodology was successfully applied as addition to procedural and object oriented programming languages, such as Java [13, 14] and C++. Cross-cutting does however also effect applications that are developed using service composition languages and methodologies. Consequently, integrating AOP principles into service composition promises a lot of potential to further improve the efficiency of application development.

This thesis explores the possibilities in particular for telecommunication service environments. Telecommunication service environments need to implement many functional and non-functional requirements that constitute cross-cutting concerns. Finding an efficient way for modularization promises therefore a particularly high gain for telecommunication service environments with direct effect on the business results of telecommunication service providers and their competitive position.

1.5 Motivation

The work on aspect orientation for composite services was started, when facing the question of how service quality metrics can be applied, monitored and managed within composite service applications. This is highly relevant in the context of service level agreements (SLA), where agreed service quality metrics are often an integral part of the contract. SLA metrics specify, how a service is expected to function. They constitutes requirements that directly translate into technical concerns for the service implementation. For example, the service level agreement might constitute an obligation for the service provider to keep the execution latency of a composite application below an agreed limit, or the accessibility of a service needs to fulfill agreed standards. Another example are media streaming applications, where content delivery needs to meet a well defined minimum quality level.

Due to exceptional situations, such as hardware failure or unexpected system load, it is not always possible to fulfill agreed metrics. For the case that this happens, the contract often defines financial penalties for the service provider. It is therefore essential to first of all know the metrics of a service delivery, so that costly problems can be detected early. As a second level of improvement, it would also be beneficial to actively control and assure that the overall metrics are kept within agreed boundaries. This might include the automatic selection and deployment of countermeasures, once some metrics are found to degrade outsides agreed limits.

Telecommunication services are often governed by a number of non-functional concerns and their related service metrics for performance, level of quality and user experience. Examples are a maximally allowed execution latency or functional concerns, such as the collection of user activity data for charging purposes. These concerns might well originate from Service Level Agreements, but there are also other sources of requirements. They frequently originate from a functional business necessity, such as the need to charge users for their consumed services or a marketing driven need for a certain level of user experience. Also legal obligations and regulatory requirements can be a source of technical concerns for applications.

Controlling the service level agreement metrics of a composite service application can be a complex task. Just measuring the parameters of the overall service delivery and usage might be straightforward, but it gets complex in case of missing the target level of quality. First of all the cause of the failure needs to be identified. In case of a composition, the cause is often an under-performing constituent service. Next to the SLA controlled relationship between the service provider and the user of the overall composite application,

the particular role of a constituent service might also be determined by an SLA. This constitutes a multi-tier and multi-layer hierarchical business relationship controlled by several independent SLA and related metrics.

In order to optimize the overall fulfillment of SLA, it would be beneficial, if the constituent services could be controlled so that local problems in some constituent services can be detected and actively compensated by other constituent services within the same composite service session. This could, for example, be achieved by temporarily selecting better performing and more expensive variants of a constituent service, or by parameterizing the used constituent services in order to reach a better performance when needed. In this scenario, a key function is the management of constituent services with respect to their individual SLA within the context of the overall SLA, as agreed with the user of the composite service application. The solution consist of monitoring if a constituent service fulfills the requirements and some logic to select the right countermeasures using the remaining constituent services being composed into the same composite application instance. As a reaction to an under-performing constituent service instance in one part of the composition better performing but potentially more expensive variants of constituent services could be used in order to compensate, and therefore still meet the overall SLA.

The solution that was proposed for handling SLA metrics within composite applications includes:

1. Determine the service metrics values to be guaranteed for the instance of the overall composite application,

2. Determine execution characteristics required from each individual constituent service in order to support the overall metrics,

3. Influence the selection of constituent services in order to choose those, which are known to deliver an execution behavior according to the SLA targets,

4. Monitor the constituent service execution in order to verify if individual and overall metrics are within desired boundaries,

5. Adapt the service selection in order to compensate for a constituent services that did not meet the targets earlier in the same session.

These are supplementary features, which need to be added to a composite application for reaching pro-active SLA control. Furthermore, these SLA management functions constitute real-time monitoring of constituent services and the features they contribute to the composition, instantaneous reflection about performance targets and compensatory actions for changing the behavior of the application. From implementation point of view, these features would inevitably cross-cut the underlying composition.

When starting to develop a framework for handling the described functions within the execution platform, Aspect Oriented Programming was not considered at first. However, the resulting implementation of a framework for reflexion and run time modification did

show high similarities to AOP variants with a dynamic online weaving. Aspect Oriented Programming (AOP) is a proven technique, which was specifically designed for dealing with this type of cross-cutting concerns. Choosing AOP as base of this dissertation generalizes and formalizes early practical considerations and solutions. This marks a natural step of anchoring the research work as contribution within a highly relevant and actively discussed branch of computer science.

Aspect Oriented Programming was already available as add-on to the Java programming language, which was used to implement the Ericsson Composition Engine and constituent services. However, cross-cutting was found to show directly on composition level rather than in the implementation of constituent services. Consequently, the definition of the composite service application itself would become tangled and scattered with a cross-cutting implementation of the needed SLA control feature. I can also be predicted that SLA monitoring and compensation handling would only one out of many potentially cross-cutting concerns once the composition engine is used in productive service environments. Thus, the definition of composite services would become increasingly complex. Consequently, composition development could loose its lean and rapid character, and with it one of its most attractive features. Therefore, this dissertation, and the AOP mechanisms it proposes, delivers a decisive contribution to keep service composition a relevant technique meeting the business goals of digital service providers.

Service performance control in the context of SLA can be essential for telecommunication, but there are many other cross-cutting concerns in that domain. Furthermore, Aspect Oriented Programming based implementation provides another important property: It allows to added or change features of the application dynamically by adding or removing aspects at run time. This potentially enables a number of use cases: The application could, for example, be customized to a particular user's needs. This would directly support long-tail business. Another example, where this ability would improve management of applications, is fast development and deployment of error corrections or policy changes. Respective measures typically effect a high number of service applications and demand rapid deployment. Implementing them as aspects, utilizing online weaving, would enable fast roll out of modifications into a broad range of composite applications at once. Especially error corrections and counter measures to security threats demand fast reaction times and AOP might be of great help achieving this.

1.6 Problem Statements

A software environment, in which applications are assembled from independent service components, fully depends on the availability of such components. A key property of these services would be that they are lean in the sense that they focus entirely on their main function. On the contrary, multi-functional services are highly specialized from composition point of view. They might only fit into applications that need or tolerate all the provided features. Therefore, complex components are less likely to match a composite application's exact requirement, and therefore, they have smaller potential to be re-used

in many application contexts. Cross-cutting concerns avoid to further split services into smaller functional modules or they increase the dependency between the modules. This leads to the first problem:

> *Cross-cutting concerns cause multi-functional service components with smaller chance to meet the exact requirements of various composite application contexts.*

The result is that SOA methodology and in particular the use of service composition becomes less efficient for the business of a service provider.

This problem is particularly evident for telecommunication network operators. Their service offer is usually accompanied by a number of requirements and concerns which introduce a layer of supplementary functions on top of the service's core function. Examples are charging of the service usage, the observation and management of quality of service or simply the logging of user activity. These supplementary functions are not strictly needed for the core service to be useful, but they are crucial for operating them within the service providers business context. These supplementary functions are frequently cross-cutting by nature. Implementing them in a conventional way on top of the core features of an application or within constituent services would mean to burden each implementation module with an increasing number of functions. The second and third problems are therefore:

> *Telecommunication services are subject to many requirements that constitute cross-cutting concerns.*

and consequently:

> *A tangled and scattered implementation is hard to avoid when addressing typical mandatory supplementary functions of telecommunication services.*

Creating and maintaining the applications, which are the result of this kind of implementation, would be complex, time-consuming and error-prone.

The existence of these requirements did make telecommunication service providers look for suitable techniques for reaching clearly structured and modularized implementations even before service oriented architectures were developed. Good examples are the Erlang [15, 16] programming language with its highly scalable component model or the architecture of the IN/CAMEL [17] service environment.

SOA was only slowly embraced in telecommunication, because of a high existing base of legacy services that were partly incompatible with SOA principles. The two biggest challenges were a high demand for controlling real-time capabilities and the complex and state-aware role of the service itself. Nevertheless, SOA became more and more adopted also in the telecommunication domain due to the advantages it beings for creating a flexible and cost efficient service portfolio.

If supplementary functions could be implemented into separate modules this would help keeping the single constituent services lean by creating more constituent services that are ideally specialized to perform a single task each. These separate modules could then be

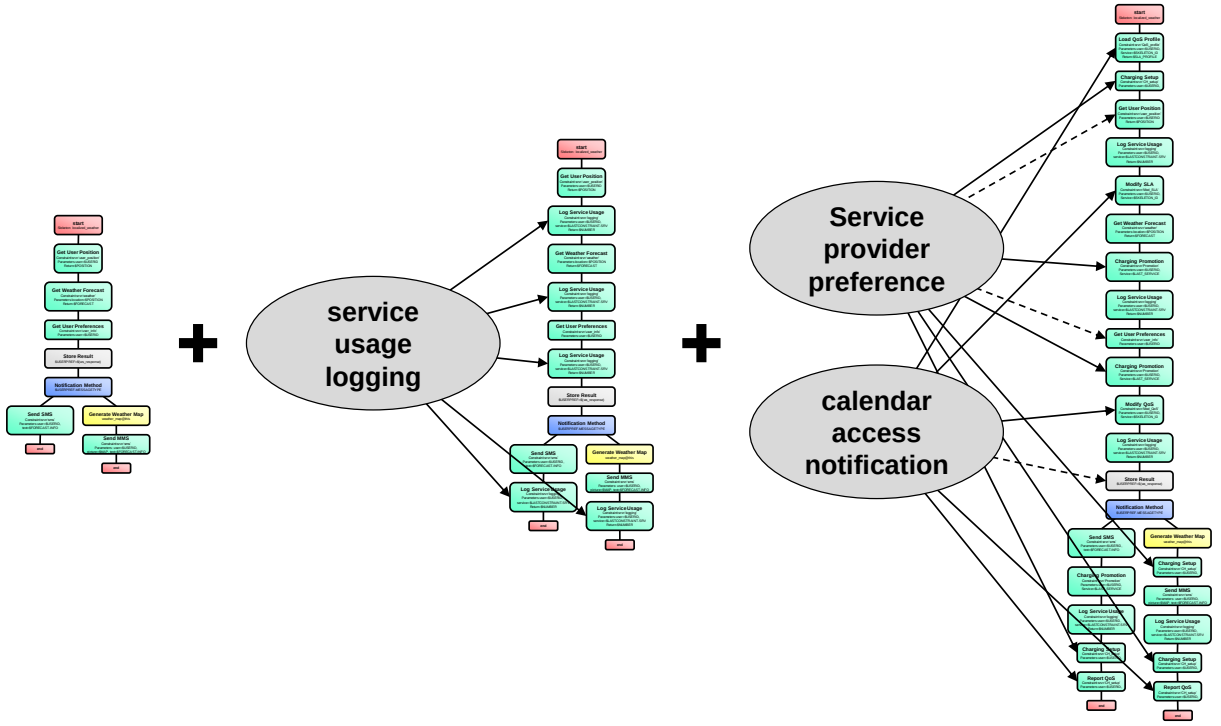

Figure 1.2: Increase of complexity when implementing additional concerns

composed into a service application, which contains all needed functionality and fulfills all requirements. Unfortunately, cross-cutting also exists on composition level and not only within the constituent service implementation. These are concerns that are cross-cutting the entire application. A simple example for this is charging. Whenever a user initiates an activity or whenever a component provides it's service to the user, the charging function needs to record the activity. A single and independent component for charging data collection first of all keeps this concern out of the other constituent services. Nevertheless, it is not possible to just solve charging by adding this charging constituent service to the composition once. It has to be connected to all potential user actions that could be charged for. Consequently, an instance of the charging component would need to be invoked at various locations throughout the composition.

Figure 1.2 demonstrates how the implementation of additional concerns is done my adding further components to a composite application. This process transforms a simple composite application with an initially lean and easy core function into an increasingly complex one. Therefore, using composition techniques did not solve cross-cutting. This means, that cross-cutting concerns not only limit the composability of constituent services, but they also lead to higher complexity on the composition level itself. The fourth problem is therefore:

Service composition methodology is directly subject to cross-cutting.

A telecommunication service application usually has a long life-cycle and needs to be maintained for many months or even years. Many people are involved over time. Error corrections will be needed and additional functions might need to be added as demands change or standards evolve. It is therefore highly important to keep the implementation as comprehensible, and thus, maintainable as possible. In this context it would be beneficial if a complex code-base with many inter-dependencies could be avoided. Every modification would potentially be complex, and therefore a costly and error-prone effort. This combined with the previous problems leads to the last problem statement:

> *High initial and operational cost are a direct consequence of typical supplementary service provider requirements especially in the telecommunication domain.*

This thesis investigates how Aspect Oriented Programming addresses these problems and how it can therefore contribute be to improvements to the business of telecommunication service providers.

1.7 Hypothesis

This thesis addresses the problems stated in Chapter 1.6. The basic idea for solving these problems is to create a service environment where suitable Aspect Oriented Programming techniques are available directly within the service composition language and methodology. A respective join-point model needs to be defined together with an aspect weaving and execution model. These are the AOP concepts that will be developed based on the targeted programming language, development model and execution semantics of the underlying composition engine. This would provide new tools to the designer of a composite application that can be used to keep implementations lean and modular even when facing multiple cross-cutting concerns.

It is therefore the primary target of this thesis to show that a service composition methodology enriched with Aspect Oriented Programming concepts can be reached and most importantly that its result is practically useful for developing composite service applications. The hypothesis related to this overall goal is:

Thesis 1
Aspect Oriented Programming concepts are a useful add on to service composition techniques

This thesis particularly focuses on telecommunication services and the special requirements of their development and execution environments. First of all this means that a base composition methodology is chosen, that is already aware of telecommunication services and able to handle their specific behavior and roles. It needs to be shown that a combination of service composition and Aspect Oriented Programming would be able to improve the business of the service provider, without causing serious problems to mandatory requirements of the domain. The second hypothesis reflects this special focus on services from the telecommunication domain:

Thesis 2
Aspect Oriented Programming can be efficiently used in telecommunication service environments

A particular common concern for telecommunication services is their real-time characteristics. This is in particular important for services that are used within end-to-end telecommunication service sessions. Their operation is governed by severe run time requirements for service applications regarding execution latency and real-time responses. Most of the quality of experience KPI of telecommunication service usage are directly or indirectly related to these real-time characteristics. The use of AOP within composition for these services must not lead to worse user experience. The second hypothesis directly addresses this.

The two hypothesis address the two central questions when introducing a new methodology for software development: Does it help the designer to be more efficient and can the domain specific requirements still be met?

1.8 Aim and Scope

This work aims at supporting the development and operation of telecommunication services. The proposed features would allow a developer to use Aspect Oriented Programming when implementing a composite service application. The special focus on the telecommunication services domain first of all demands the choice of a suitable composition technology as base for developing the additions for Aspect Oriented Programming. For this reason the Ericsson Composition Engine and its related composition language and execution semantics are selected as base for the AOP solution. The thesis will develop a solution concept based on this composition technology.

With the Ericsson Composition Engine a proprietary technology is chosen as base of this thesis. It is a service composition technology that contains a set of capabilities that enable it to operate directly with the special role of services within telecommunication sessions. Composition environments based on the widely used BPEL and BPMN lack these capabilities to a great extent. Nevertheless, this choice potentially reduces the general application of the results presented in this thesis. However, the Ericsson Composition Engine is to a great extent designed as a general purpose composition engine with many similarities to openly available and standardized technologies based on BPEL and BPMN. The differences are described in detail in this thesis. The telecommunication service capabilities are realized to a great extent as additional features to a general composition methodology. The addition of the AOP concept is mainly based on these general service composition concepts, which makes the results easily transferable to other service composition environments.

The proof of concept implementation, which is done as part of this thesis, is using general purpose programming techniques and concepts. This potentially allows transferring the work to a similar implementation based on a different composition engine. It could be extended with the respective features for Aspect Oriented Programming support. With that, also the qualitative discussion, for example on composition efficiency, could also be

transferred. For this reason the results of this thesis have an extensive general validity regardless of the choice of a proprietary composition engine.

In order to address thesis 2, the run time performance of the AOP enhanced composition engine gets high attention and is assessed in detail for telecommunication and non-telecommunication use cases.

A special potential capability of AOP techniques can be an overall improved service handling rather than just a more modular service implementation. This refers to flexible modifications to applications at run time. This would create unique abilities for management and maintenance of composite services throughout their life-cycle. These use cases promise a considerable gain for the operation of service environments, and for this reason their discussion is in scope.

It is a clear target of this thesis to provide valuable input to potential commercial implementations of a composition engine. It is in this respect essential to understand what AOP features would be needed for typical use-cases and what performance can be expected. In order to address this the thesis provides an overview of the extent to which a certain AOP feature with its respective implementation in the composition engine contributes to the overall performance.

1.9 Research Questions and Methodology

There is already a number of different AOP solutions available. Most prominently there is AspectJ [13], an AOP addition to the Java programming language. Their unique characteristics were designed and evaluated in order to support a wide range of use cases. Previous research has explored their strengths and weaknesses as well as their main characteristics and performance. The first research question would therefore be what properties the wanted AOP solution for service composition should have if it would be specifically designed for service composition in telecommunication. In order to address this question, the thesis provides an assessment of previous work in the field of Aspect Oriented Programming and service composition. Known techniques are assessed against the targets of this thesis and potential gaps are identified leading to a feature outline of the specific solution that is chosen to be developed.

Based on the wanted characteristics of the AOP solution, the next research question would be finding the conceptual details of the proposed solution. Several distinct questions are addressed that are typical for the introduction of an AOP methodology to an existing programming language. What would, for example, be a suitable join-point model? What would be a good weaving model and language? How would advice be represented? How are aspects deployed and executed? Answers to these questions are found through an analysis of the target composition language and environment combined with the experience from earlier cases of introduction of AOP to an existing programming language.

The proposed concept is verified against the two hypothesis presented in Chapter 1.7. The verification is facilitated by a reference implementation of the proposed AOP solution concept based on the Ericsson Composition Engine. This creates a fully operational

service composition execution engine that contains the proposed capabilities for Aspect Oriented Programming. Experiments with the reference implementation provide a deeper understanding of the proposed solution's capabilities and its run time behavior.

The implementation of typical use cases based on the added AOP features is demonstrated. Here a comparison between conventional and AOP based implementation allows reasoning about the advantages of the proposed solution in terms of development efficiency.

A comparison of the composition performance with and without using AOP is an essential part of the verification. It particularly shows if and under which circumstances the AOP features can be used without violating the performance characteristics of telecommunication services. Furthermore, the experiments with the prototype allow creating a mathematical model of the execution engine. It distinguishes and quantifies the latencies caused by various distinct parts of the implementation.

The mathematical model of the run time behavior of the composition engine is the base for further exploration of implementation variants. This answers the question, if and to what extent the use of certain AOP features and capabilities would influence overall performance of a service application. Based on this analysis recommendations are provided that first of all address the question if the AOP solution is ready for use in telecommunication service scenarios. It also provides indications for what might be the best balance between features and performance for a specific application scenario. The goal of this effort would be to provide guidelines for the design of a commercial composition engine with well defined performance characteristics and features.

1.10 Contributions

This work pioneers in the deployment of Aspect Oriented Programming techniques for dynamic service composition. It advances the understanding of this domain through the following unique contributions:

How to design AOP Languages:

The full design process of a new language for aspect oriented programming is documented. This starts with an analysis of the unique requirements of the application domain and the available languages for application development. It also includes all reasoning about AOP functions and capabilities. In this step the weaving method is selected, a suitable join-point model is chosen and language extensions are proposed for expressing weaving instructions and for representing advice. The full reasoning process about all detailed design decisions of the new AOP methodology is documented. Thus, this dissertation can be used as a template for designing AOP methodology for any base language and application domain.

Online Weaving Methodology and Languages:

The use of Aspect Oriented Programming techniques in general purpose programming languages and frameworks was already established and well understood. However, the vast

majority of solutions focuses entirely on offline use of AOP. There is very little research on online waving. This dissertation contributes a unique Aspect Oriented Programming language, which enables the dynamic weaving of aspects into application instances at run time. This marks a major advancement in the understanding of online weaving.

Aspect Oriented Programming for Composite Applications:

First proposals for adding AOP to business process languages, such as BPEL, were already known, but the opportunities of AOP in this domain were far from being fully explored. This dissertation marks a major advancement of the domain. It contributes a fully integrated and fully functional AOP language and development environment, addressing specific challenges of composite service development and management. The solution is complete to the extent that it would enable using the described methodology, the developed language and the implemented tools within productive development of composite applications. The focus on practicality of proposals and completeness of the solution is a major strength of this dissertation.

Dynamic Composite Applications in the Telecommunication Domain:

The composition methodology and language used as base of this dissertation is characterized by dynamic service selection at run time and with heterogeneous constituent services. Both features are not common for standard SOA based composition design. Nevertheless, they have high relevance for telecommunication use cases. The dissertation contributes an AOP solution with respect to these capabilities.

Furthermore, this dissertation proposes and verifies the use of AOP and service composition combined in the domain of telecommunication service applications. This context imposes unique requirements on applications, which are specifically addressed. The most important property is the run time performance of composite service application within an end-to-end usage session context. The central contribution from this angle is an in depth assessment and discussion of AOP feature implementation alternatives with respect to performance.

Reference Implementation of Online Weaving:

The dissertation shows an example how dynamic weaving can be implemented as extension of a composition execution engine. Major design challenges and their solution are described. For example, the dissertation shows how to use event handling mechanisms for creating an online weaving monitoring and advice invocation overlay. It furthermore discusses the practical implications and the proposed solution for potentially conflicting access to run time session data. The reference implementation embodies these solutions, but this contribution is widely universal. The detailed solutions can easily be utilized as base for other implementations of AOP enabled composition execution engines.

Characteristics of Online Weaving:

The composition and weaving performance is a particularly important challenge of telecommunication being the primarily addressed application domain of this dissertation. Nevertheless, execution performance is a universally important concern especially for online weaving. This dissertation contributes a detailed assessment of overall execution performance of composite applications. It facilitates a discussion, if the proposed solution and the related conceptual design decisions meet universal and domain specific requirements. The contributed detail level allows to distinguish application use cases, for which AOP can be applied from those, where it should be avoided from performance point of view.

The performance evaluation also contributes a mathematical model of the reference implementation. This dissertation demonstrates how a performance model can be obtained and parameterized from measurements. This model allows concrete predictions of AOP performance for specific implementation variants of the execution engine. Valuable insights can eb obtained, if and under which circumstances the use of AOP is suitable for applications within the targeted domain. Not only the reference implementation is discussed, but also concrete performance figures for variations of the implementation can be derived from the model.

Based on this model, functional variants in the AOP concept can be verified individually and in combination. This provides an overview of different AOP solutions and features and their individual impact on performance. The contribution of this dissertation is, in this respect, more than a verification, if the proposed solution works for a specific set of requirements. This dissertation rather contributes a universal quantitative performance model, which allows finding a working trade-off between AOP features and composition performance. This is a highly valuable contribution for deciding the design details, when implementing future AOP enabled execution engines. Future commercial implementations of an AOP enhanced composition engine would gain very valuable insights form this dissertation about expected capabilities and characteristics.

Dynamic Management of Concerns:

Aspect Oriented Programming is typically proposed as an implementation tool. It supports the developer in efficient treatment of cross-cutting concerns. This dissertation pioneers in an extended use of Aspect Oriented Programming concepts. A major challenge for digital service providers is the management of service applications. Next to the directly addressed telecommunication, other emerging domains, such as pervasive computing [18], cyber-physical systems and the Internet of Things [19] demand contextual adaptation of application behavior. Security and fraud prevention also requires rapid responses.

The dissertation explores, how dynamic weaving facilitates a flexible management of aspects and concerns in order to address these domains and use cases. It discusses how the life-cycle of an application and aspects are related to each other and how a dynamic automatic assignment of aspect could be reached through rules engines. The AOP proposed solution is discussed as a sound foundation for dynamic aspect management based on

rules and policies. Although these topics are not the central focus of this dissertation, it nevertheless contributes a base for further research.

1.11 Structure of the Dissertation

A general overview of the main concepts and solutions of aspect orientated programming and service composition is provided in Chapter 2. It also includes an overview of the central concepts of telecommunication services and the telecommunication domain in general with its specific service usage models and requirements. In this respect the Ericsson Composition Engine is introduced in detail, because it is the service composition technique of choice for this thesis.

Chapter 3 introduces a service development process and in particular a model for service application life-cycles. A special focus on the role of aspects within the life-cycle of composite service applications. This leads to the idea of flexibly applied aspects, their partly independent life-cycle and a discussion of related challenges.

Chapter 4 explains the details of the proposed solution based on an analysis of the requirements and gaps in today's solutions. These considerations lead to the details of the proposed AOP solution, for example the selection of a join-point model, the definition of a weaving language and a descriptions of how to develop and apply aspects with the proposed methodology. This is accompanied by an architectural proposal of how the AOP functionality and in particular the aspect weaving can be integrated into the existing composition engine.

The implementation of the proposed AOP solution is verified and the results are presented in Chapter 5. The functional capabilities of the AOP solution are demonstrated based on typical use cases. Traditional and AOP based implementations are compared in order to find out whether using AOP brings advantages and if the proposed AOP solution therefore provides useful tooling to the composition developer.

Furthermore, in Chapter 5 the performance of the AOP solution is measured based on a reference implementation. These measurements allow to create a quantitative mathematical model of execution time for a composite service application. In particular, this model identifies the major contributions to composition latency based on the reference implementation. It distinguishes the contributions of the basic composition and the major functions of the aspect weaving solution. This model is the base of a quantitative discussion of implementation variants and their expected performance. This leads to concrete recommendations of what AOP features are feasible for different types of service application scenarios.

The thesis concludes in Chapter 6 with an overview of major findings, recommendations and proposed future work.

Chapter 2

Background & Related Work

This section introduces related and preceding work that is important as foundation for this thesis. There are two main threads of work this thesis is bringing together: The first is service composition and more specifically a service composition technology that was developed based on the unique requirements and service usage models of telecommunication networks. On this background also service usage in telecommunication networks is briefly introduced. This points out the differences between telecommunication services and other well established technologies in service science. This leads to the Ericsson Composition Engine. It is a unique orchestration solution for telecommunication and the base of this thesis. Thus, an in-depth introduction will be provided.

The second main technological thread is Aspect Oriented Programming. This section will provide a brief overview of the solved problems, the main concepts, and variations of this technology. Also the specific terminology of this domain is explained.

2.1 Telecommunication Networks

Since the first telephone was invented and demonstrated in 1876 [20] and until long into the late 20th century, the main task of a telecommunication network was to provide an efficient voice connection between the telephony equipment of two users. Thus, the end-to-end transmission of speech was the predominant service. One of the central principles of telecommunication networks since these early days is the call path.

Although its name refers to voice telephony, the concept as such is independent of the transmitted content and still dominates the thinking and design principles of telecommunication solutions until today. The task of the telecommunication network is to establish paths as direct links between pairs of user equipment at both ends across potentially long distances and for a substantial number of participants. Figure 2.1 shows the concept of an established call path across a telecommunication network.

A couple of basic requirements are essential for a practical feasibility and quality of experience. These requirements govern the telecommunication architectures and technical solutions:

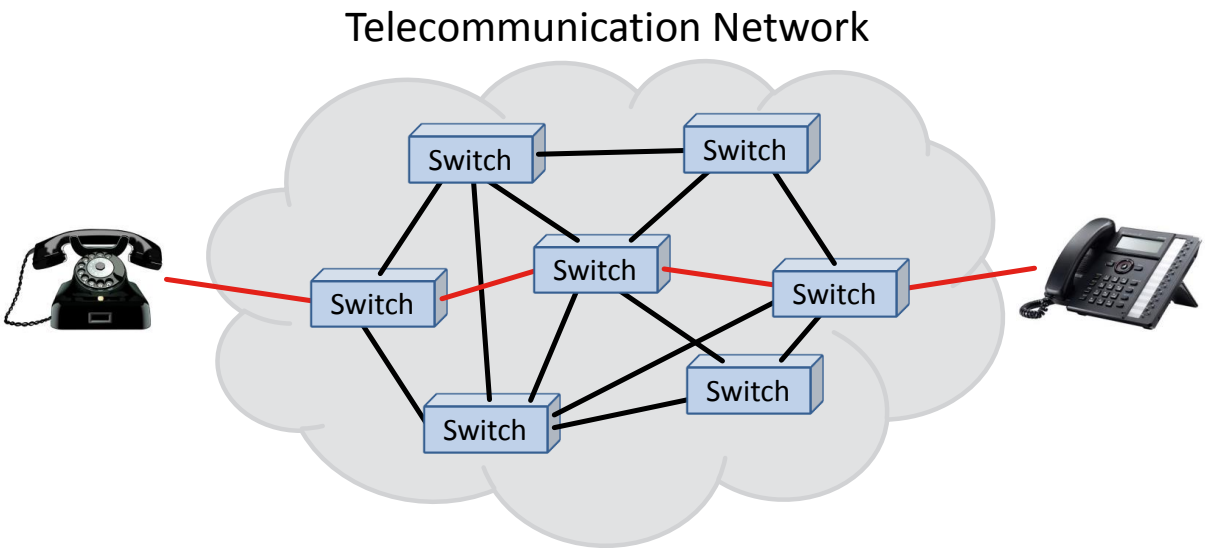

Figure 2.1: Basic call/session path in telecommunication networks

- **Setup Latency:** The time to establish the connection should be short in order to avoid long waiting until the other participant is reached. A setup time of up to a few seconds was for a long time considered to be acceptable especially with voice between human users as the main service. However today, with new services, such as interactive gaming and entertainment, push-to-talk communication and instant messaging, much shorter setup times in the magnitude of a few 100ms might be required for good service experience.

- **In Service Latency:** The latency of the transmitted content needs to be short in order to allow an intuitive way of communication. It is highly disturbing for a human user if the transmission of a user's voice to the user on the other end and in return the answer needs a lot of time. Latencies of up to 200 ms are usually considered to be still ok for voice services. However, also here modern applications might have much more demanding requirements.

- **Scalability:** Both latencies must not degrade, even if the network is large by number of users and consequently the number of parallel sessions increases.

Technical solutions for telecommunication have to deal with severe real-time requirements, because low latency is a key property of intuitive end-to-end voice service. This can be reached relatively easy for a few telephones within a local area, but the networks did grow and the solutions needed to scale to millions of users and across hierarchically organized local, national and international networks. The concept of a call path became essential for meeting the requirements at scale, with the technology at hand and with as good utilization of the infrastructure as possible in order to bring costs of service down.

2.1.1 The Call Path and Supplementary Services

Circuit switching was for a long time the technological base of telecommunication networks and the solution for meeting the major latency and scalability requirements. It refers to exclusive reservation of network resources for each end-to-end path between user equipment at the end-points. The resources are tied to the service, e.g. a voice call, for the entire duration of the service usage. In the beginning, the call path was set-up using exclusive analog lines that were connected manually using manual switchboards and later this was automatized using electro-mechanical or electronic switches and basic in-line signaling protocols. Later, this was replaced by reserving channels on a shared physical medium using time devision multiplex (TDM) and dedicated signaling networks.

Regardless of these technological advances, the basic idea of an end-to-end path is kept for a good reason. Once established, it only needs minimal supervision and data processing. All routing decisions are, for example, taken while establishing the end-to-end path by reserving all needed physical resources. As a result, the processing effort and all routing decisions are mainly in the setup phase with its more relaxed end-to-end latency requirements of several seconds. Once it is in place the data can flow along the pre-reserved path without any major intermediate processing needed. This way the much more severe requirements for in-service latency of below 200 ms could be met. This ultimately allowed the users to naturally speak to each other.

For a long time the described call path based solution was considered to be the only feasible way to fulfill these real-time requirements with the technology at hand. This focus on controlling real-time requirements and related service quality in order to reach a particular user experience at high scale can historically be considered to be the main separating factor of telecommunication from other network technologies. For example, early IP based networks were designed for tolerating partial network outages. This is reached by taking separate routing decisions for each single packet of data. Thus, all processing for routing along the packet path is done in-service. This resulted in a need for more relaxed latency requirements and the typical best effort approach to service usage latency.

Improvements in telecommunication networks were done mainly in scaling the size of the network up while reducing the costs per user leading to better utilization of resources and more efficient control and routing solutions. With the introduction of digital exchanges in the 1970s the call path setup and control became mostly software driven. This enabled increased flexibility and shorter development cycles. It also became the starting point of a constantly increasing number of supplementary services because in many cases a new service could be realized entirely in software. Examples of supplementary services are call forwarding, number translation, call barring or conference calls.

For supplementary services the call path concept played also an important role. They were allocated along the end-to-end signaling path and they mainly implemented their function through listening to, intercepting and changing the end-to-end signaling. In the call path establishment originating at the calling user the path setup signaling passes various service entities. Each of them might modify, for example, the destination address

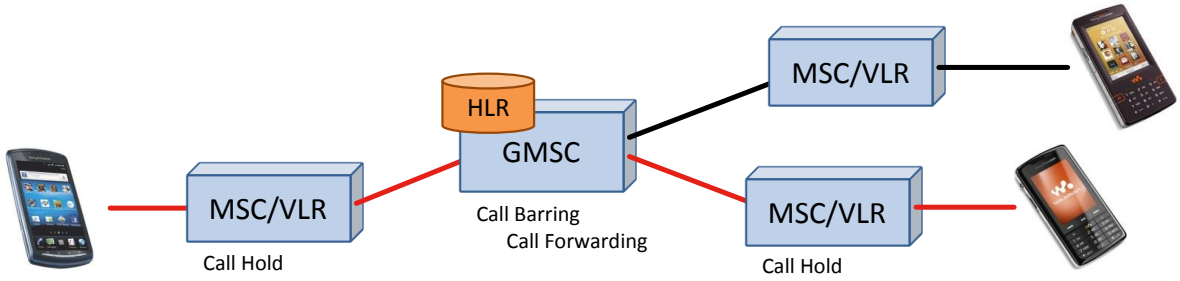

Figure 2.2: Allocation of supplementary services along the call path in the GSM mobile core network

or otherwise check and alter the call path setup. Also, after establishment of the call, supplementary services participate in the call related signaling. They might decide to apply changes, such as releasing the session or re-routing a part of the call towards a new destination.

It is important to note that the supplementary services are relatively self contained and usually state-aware units. They became modular additions to the call control and they interact according to well defined protocols and signaling procedures. In order to make this work, elaborate signaling standards guaranteed compatible service behavior across the implementations of several vendors.

All the statements above in this chapter are not only valid for fixed line digital telecommunication networks, but also for circuit switched digital cellular networks. They are, for example, based on GSM and similar standards and usually referred to as 2G. On high level, the main difference between cellular and fixed networks is that mobile core networks need extended routing functions that are aware of the location of the participating end devices in order to route the call to the respective cell. The call path concept itself and the roles of supplementary services along the path are in principle the same.

One important technological development in telecommunication is the separation of control from the actual transmission of the payload in a layered architecture. The signaling infrastructure for establishing a call path and the controlling nodes, such as the Mobile services Switching Center (MSC-Server), reside on the control layer. In this architecture, the call path refers to the logical signaling path through the control layer. Protocols, such as the ISDN user part (ISUP) and Bearer Independent Call Control (BICC), are used to control it. The payload for signaling and user services are transmitted in the user plane. The setup of respective network paths in the user plane is controlled by the nodes of the control plane and their signaling.

The transition from a circuit switched to packet switched technology was done by deploying a packet switched user plane. The first solutions were based on Asynchronous Transfer Mode (ATP) and soon after Internet Protocol (IP) based solutions followed. Although the user plane became packet switched and uses IP technologies for data transport, in the control plane the notion of end-to-end paths, and thus, the role of services within

the overall session control is still valid. Figure 2.2 shows a path through typical network nodes of a GSM mobile core network. The figure also shows for typical services, such as call hold, call barring and call forwarding, in which nodes along the call path their service logic is allocated.

2.1.2 A Dedicated Telecommunication Service Layer

Telecommunication was always subject to detailed and extensive standardization in order to enable seamless inter-work between networks of different operators and based on equipment from various vendors. The basic network services, for example routing and setup of bearer sessions, are realized this way based on standardized signaling methods and procedures. No telecommunication network would work without them. These functions are usually hard-coded into the core network control nodes for optimal performance.

The standards also specify the detailed inter-work and signaling schema for a number of supplementary services, such as call forwarding, call barring, call hold, multi-party service, number translation, closed user group, explicit call transfer or handover of calls for roaming subscribers. These service require inter-work between several network nodes, which in general reside within multiple networks along the call path. In principle most supplementary services can be considered to be optional, but in practice they are implemented according to standards and deployed in every network. Their availability is often a prerequisite demanded by national and international regulation, or they are considered a given asset that is always available. Usually all networks provide them at relatively good quality and users take for granted that they exist. From business model point of view they are a commodity.

In order to provide a unique and attractive offer to the users, additional value-added services are used. They might also be standardized, but they are usually optional and allow the operator to distinguish itself through availability of unique functionality. Thus, these services are ideally not yet available from competitors. However, following the same implementation model used for basic and most supplementary services has a couple of disadvantages: It is time consuming and changes would need to follow the usual release and roll-out schemes of the core network software based on planned maintenance windows. This is not agile and the innovation cycles would be much too long to flexibly react on market demands. It also bears an implicit danger to the network stability and reliability. A development and roll-out process that would flexibly allow dynamic deployment of services would be needed. This was the reason for introducing Intelligent Networks (IN).

Intelligent network services are designed based on a client-server model while logically preserving the service allocation along the call path. IN introduces a service layer on top of the existing control layer. It therefore separates the implementation of value added services and also of new supplementary services from the main network control nodes. Standards define a generic and flexible interface between the layers that allow the services to interact with session control functions. In IN terminology the application servers in the service layer are called Service Control Points (SCP).

Today, IN for cellular networks is usually based on Customized Applications for Mobile

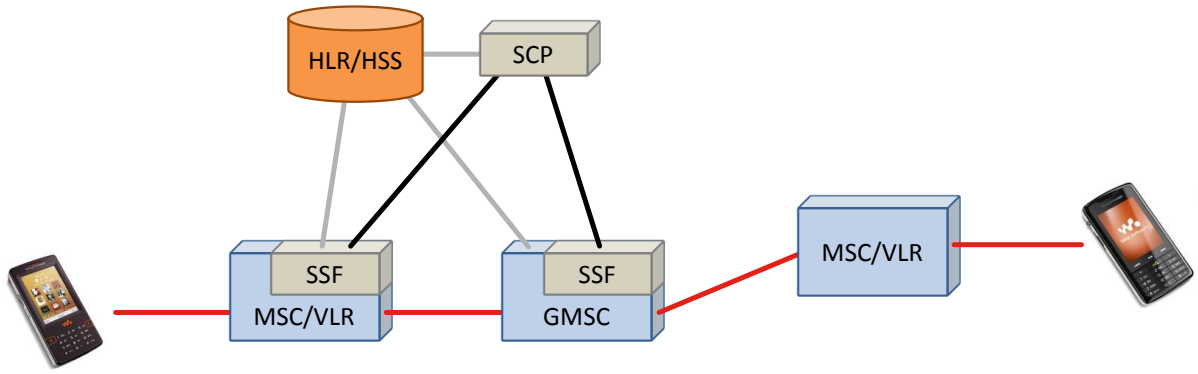

Figure 2.3: CAMEL as service layer infrastructure for mobile networks

Networks Enhanced Logic, or in short CAMEL [21, 22, 23]. CAMEL has extended IN by adding support for mobility features as needed in cellular networks [17]. IN and CAMEL allow developing services and deploying them flexibly on the SCP without impacting the network control layer implementation [24].

The SCP communicates with the Service Switching Function (SSF) that resides within the traffic control nodes and provides a set of well defined inter-work capabilities with the control functions. A well standardized protocol, the CAMEL Application Part (CAP) [25, 26] exists for this purpose. The SSF detects, for example, if a service in the SCP needs to be invoked based on session signaling and notifies the respective service that resides in the SCP using CAP. The SSF also executes the modifications to the session setup on demand from the service logic. The actual control logic of the service resides in the SCP and the influence on the call/session setup and call/session control is applied through the SSF.

Still today the IN/CAMEL infrastructure is an integral part of 2G and 3G telecommunication networks. Major services are implemented based on this architecture. Prominent examples are prepaid charging, virtual private networks or televoting. Please note, that also here the logical allocation of services on the call path has not changed. The service logic may be moved up into the service layer, but through the SSF they can reach out into the control plane and have a direct effect on session signaling and session control. The service is still a logical component in the call session path.

2.1.3 The IP Multimedia Sub-System (IMS)

The IP Multimedia Sub-System [27] constitutes the core network control layer of Next Generation Networks (NGN) [28]. IMS is standardized by the 3rd Generation Partnership Project (3GPP) [29, 30, 31] and ETSI Telecoms and Internet converged Services and Protocols for Advanced Networks (TISPAN). It is based to a large extent on standards and technologies specified by the Internet Engineering Task Force (IETF).

IMS is designed to work with many different access technologies. Examples are mobile phones in a cellular radio network, fixed line equipment or wireless LAN attached clients.

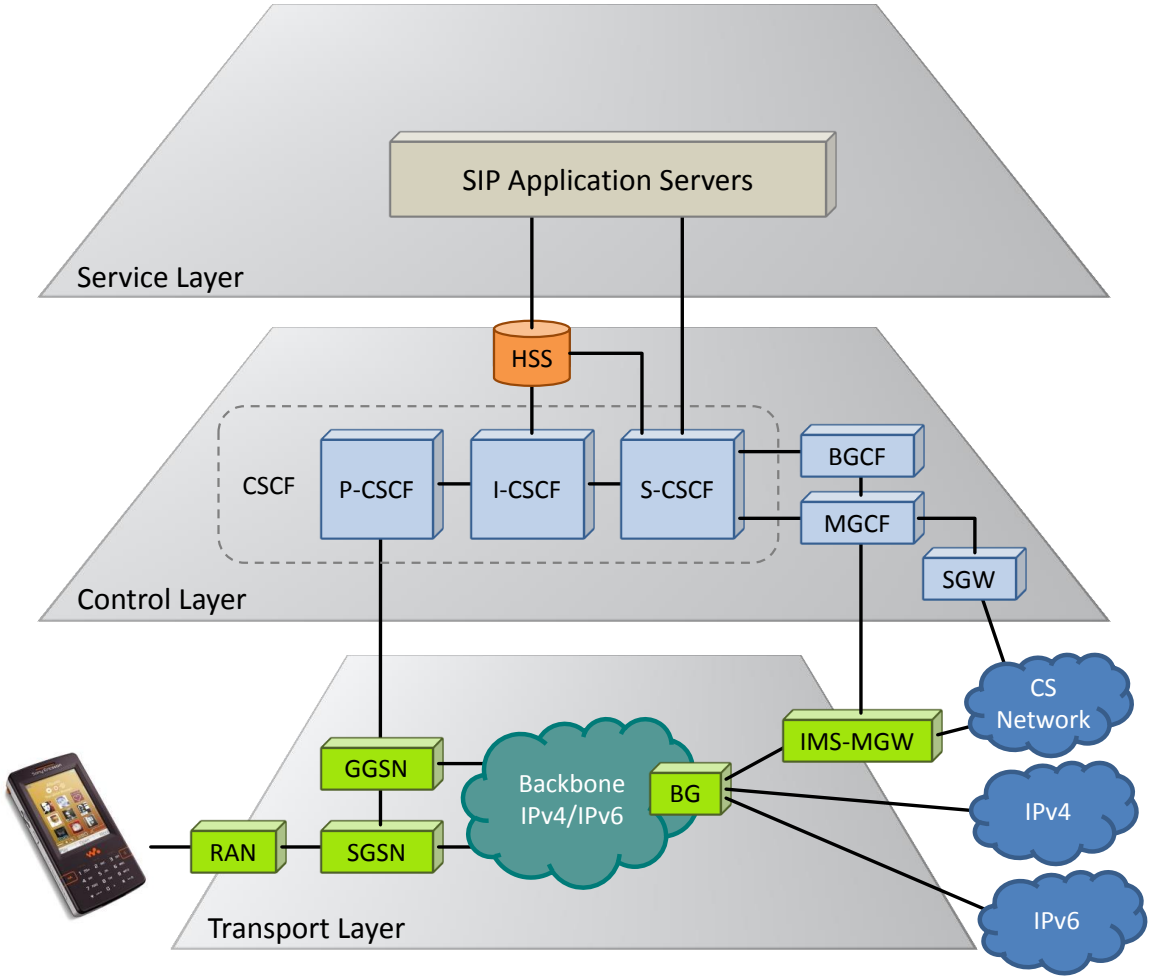

Figure 2.4: IMS architecture

They all can be served by an IMS controlled network. Figure 2.4 shows a simplified view of the IMS architecture with the most essential nodes. Here the nodes are shown that are needed for a terminal within a cellular network on one side being connected to another client that resides either within a circuit switched network or IPv4 or IPv6 based networks. Therefore, the transport layer shown in Figure 2.4 consists of a Radio Access Network (RAN) serving the mobile client. Furthermore it consists of a packet core network with the Serving GPRS Support Node (SGSN) and the Gateway GPRS Support Node (GGSN). The SGSN and GGSN are the central transport layer nodes in a packet switched mobile core network. They utilize an IP based Backbone (BB). On the other side the transport layer integrates into the transport layer of a circuit switched network through the IMS Media Gateway.

While the transport of all payload is taken care of in the transport layer, the control layer contains the logic of end-to-end session control. In IMS the protocol for end-to-end

session signaling is the Session Initiation Protocol (SIP) [32, 33]. SIP is an application layer protocol specified by the IETF that was adopted by 3GPP for IMS signaling. All SIP messages, while being carried in the transport layer, are sent up to the logic that resides in the control layer. If the attached non packet switched networks use a different signaling method, a translation of signaling messages is performed by the Signaling Gateway (SGW). It terminates both SIP and SS7 signaling that is used in legacy circuit switched networks and translates between them.

Multiple underlying transport layer protocols, such as TCP, UDP or SCTP, are possible for SIP. It is a text based protocol incorporating elements from HTTP and SMTP. In this respect it follows a request/response transaction model similar to the one of HTTP. Each transaction consists of a client request that invokes a particular method or function on the server and demands at least one response. SIP reuses most of the header fields, encoding rules and status codes of HTTP.

A central goal for SIP was to provide a signaling and call setup protocol for IP-based communications that can also support call processing and supplementary services from public switched telephone network (PSTN). SIP by itself does not define these features. Its focus is rather call session setup and signaling. It therefore enables telephony-like operations, such as dialing a number, causing a phone to ring, hearing ring-back tones or a busy signal. Furthermore, it was designed to enable network elements like designated proxy servers and user agents. SIP-enabled telephony networks can also implement many of the more advanced call processing features that before were realized using Signaling System 7 (SS7). SIP is also a peer-to-peer protocol. It therefore only requires a simple, thus, scalable core network with intelligence distributed in the network edge and embedded in communicating endpoints.

The main nodes in the IMS control layer are the three variants of the Call Session Control Function (CSCF):

P-CSCF: The Proxy CSCF is a SIP proxy and the first point of contact for an IMS terminal. It is usually located in the visited network, thus, close to the user terminal, and it provides subscriber authentication and session security towards the terminal.

S-CSCF: The Serving CSCF is the central node in IMS signaling. It is a SIP server performing session control and routing. It is always located in the home network. Using Diameter Cx and Dx interfaces to the HSS it gets user profiles and through the ISC interface it invokes and integrates services into the session.

I-CSCF: The Interrogating CSCF mainly assists the S-CSCF in the interaction with remote SIP servers. It is exposed to remote servers through DNS and it forwards SIP requests and responses to the S-CSCF.

An essential node for service routing in IMS is the Home Subscriber Server (HSS). It is a master user database containing user subscription-related information. This supports the CSCF in authentication and authorization of users and in setup and control of end-to end sessions. The HSS also provides information about the subscriber's location and services.

It therefore has a similar role as the Home Location Register (HLR) in circuit switched mobile networks.

A SIP user agent (UA) is a logical network end-point where the SIP protocol terminates. A SIP transaction consist of a user agent client that is sending a request and a user agent server that receives the request and returns a response. However, these roles are taken per transaction and might change within a control session that involves many of these transactions. A SIP phone is a user agent that provides the typical function of a telephone, such as dial, call transfer, call hold and answer.

A new SIP call session set-up is started by a user agent sending a SIP INVITE Message. This message will reach the S-CSCF, where the essential routing decisions are taken with support from the subscription data stored in the HSS. The S-CSCF routes the SIP request towards the destination endpoint, for example the mobile phone of the called subscriber. The SIP invite will then finally reach the destination user agent.

The control plane signaling path that is set-up for a call session is usually kept throughout the duration of the call. With the first forward message the session path is initially build. Subsequent SIP requests sent be user agent within the calling subscriber's client will follow that same session path in forward direction. The called client send SIP requests within the same session context. They will propagate long the same the session path in reverse direction. The result of these signaling procedures is in principle the same persistent end-to-end call path that was always typical for telecommunication networks. This concept is therefore decisive also for the service usage model of IMS as explained in more detail in Chapter 2.1.4. Any service and control node along the session path can intercept and modify the signaling. This node takes the role of a SIP application server terminating the SIP request. It sends a reply back in opposite direction or it only modifies the message and sens it on.

Please note that the above is only valid for the signaling in the control plane. The routing and paths in the transport layer are to great extent decoupled from SIP end-to-end session path in the control layer.

2.1.4 Service Routing in IMS

In IMS based networks the logic of most supplementary and value added services is implemented in a dedicated service layer. This leaves the control layer relatively lean and it constitutes a clear separation of concerns between service logic implementation on one layer and routing and session control functions on the other.

The service layer of IMS contains SIP Application Servers (SIP-AS). They host supplementary and value added services. Even many of the basic functions and supplementary services of a telecommunication network are implemented in the service layer rather than being included into the control layer nodes. Consequently, one central role of the IMS control layer nodes is incorporating the separated service logic into session control. For this purpose the IMS standards specify the IMS Service Control (ISC) interface between the CSCF and the application servers. Figure 2.5 and Figure 2.6 show how the SIP applications servers are included in the forward setup of a SIP call.

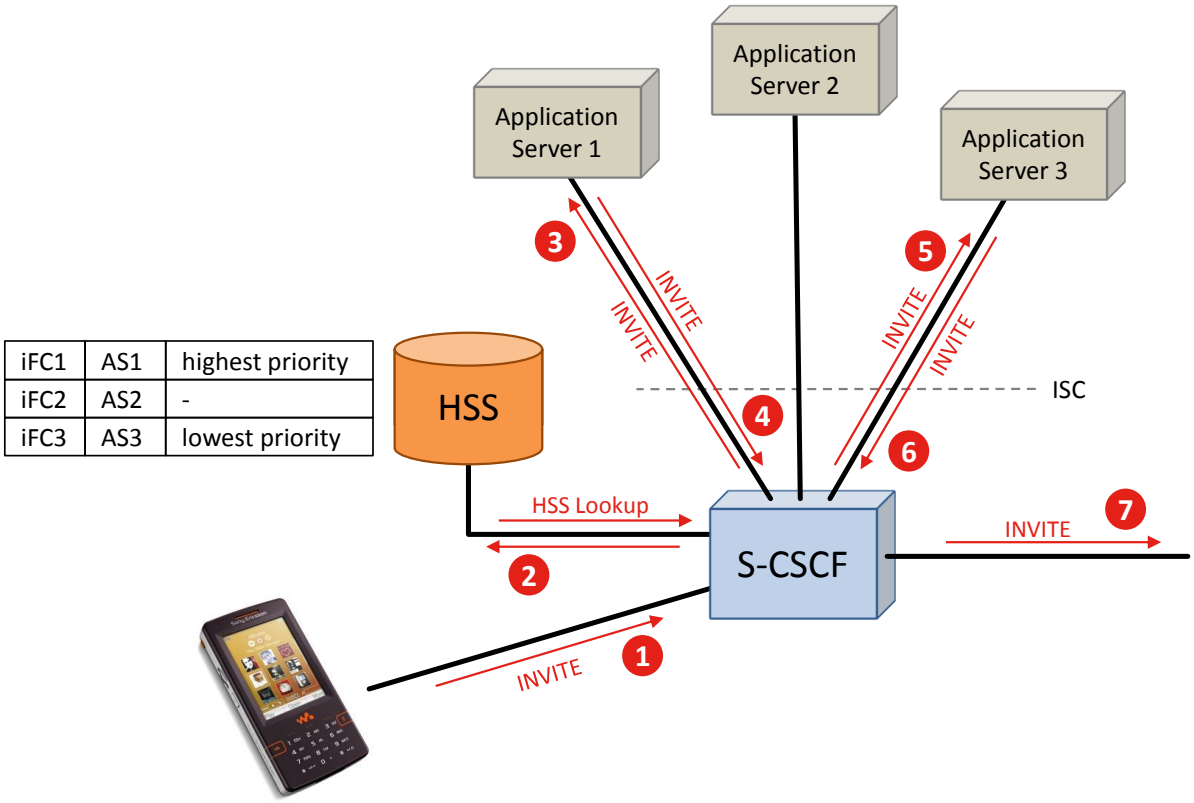

Figure 2.5: SIP forward setup with INVITE request and service routing

A SIP application server can be realized by a Java application servers equipped with a SIP protocol stack. One example of a SIP application server is Sailfin [34]. It is a Sun Glassfish Java EE Application Server [35] combined with an open source SIP protocol stack. It was developed in an open source project driven by Ericsson and Sun, before Sun was included into Oracle. Today, a similar SIP stack is available for most commercial JEE application servers.

For understanding the Ericsson Composition Engine, and therefore for this dissertation, it is important to understand the unique role that SIP services take within the context of telecommunication service sessions. The HSS subscriber profiles contain the so called initial Filter Criteria (iFC). They represent a provisioned subscription of a user to a service. Thus, they refer to all services that shall be included into a SIP session for a user.

The iFC may contain an application server URI, towards which the SIP request shall be forwarded by the S-CSCF. Thus, on reception of a SIP request the S-CSCF sends it over the ISC interface to the first SIP Application Server pointed to by the iFC. The SIP addressed SIP service on the application server processes the request and then either terminates it and answers with a reply or it forwards the SIP request back to the S-CSCF. The request might be modified before it is forwarded. A typical example of a possible modification is the change of the destination address. Features, such as call forwarding or

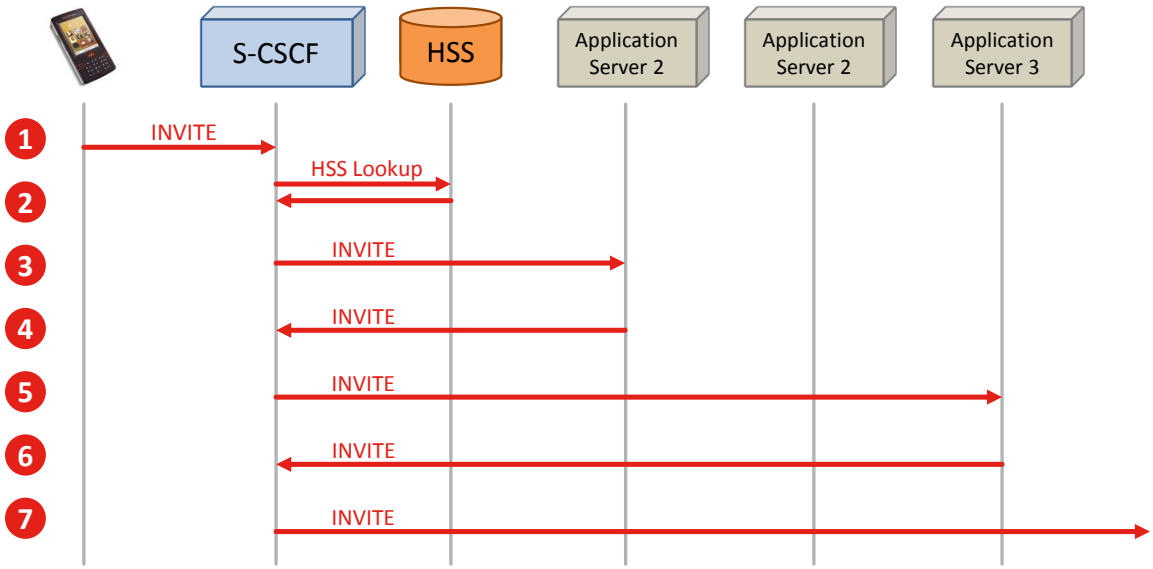

Figure 2.6: Message flow in SIP forward setup

Number translations, can be realized this way by using a SIP service for manipulating the destination of the request.

When receiving the SIP request back from the application server, the S-CSCF will repeat the procedure with the next application server and service pointed to in the user's iFC. This dialog between the CSCF and the application servers is done until all iFC are processed, and thus, all needed services are involved in the session setup. If there are no more iFC to process, the S-CSCF forwards the SIP request towards its final destination.

If no service terminates the SIP request, the entire iFC processing is based on forwarding. Also sending back the SIP request to the S-CSCF is a forward. This means a logical session/call path is built traversing all services that were addressed by iFC. Figure 2.5 shows the setup of the session call path with the S-CSCF involving multiple services. It also shows the logical path being maintained where an instance of each service stays in the signaling chain. Figure 2.6 shows the respective flow of messages between the nodes.

Once involved in the session path, a service instance listens to and evaluates all SIP signaling that is passing by. The service instance is waiting for any SIP message it needs to act upon. Being always involved, a service can decide to act and it can apply extensive modifications to the signaling. It can terminate a message and send a response. It can also modify data in the message and forward it or it can simply decide to do nothing so that the message proceeds along the signaling path untouched.

This way of operation demands that a service is aware of and understands protocol and session state. The service is also not simply acting on a request from a client generating a response. It is continuously monitoring a flow of protocol messages deciding if it needs to act. This way of service usage is typical for telecommunication networks. It can result in a high distribution of service logic and it definitely demands a high degree of collaboration and coordination between services. Standardization coordinates the behavior and allowed

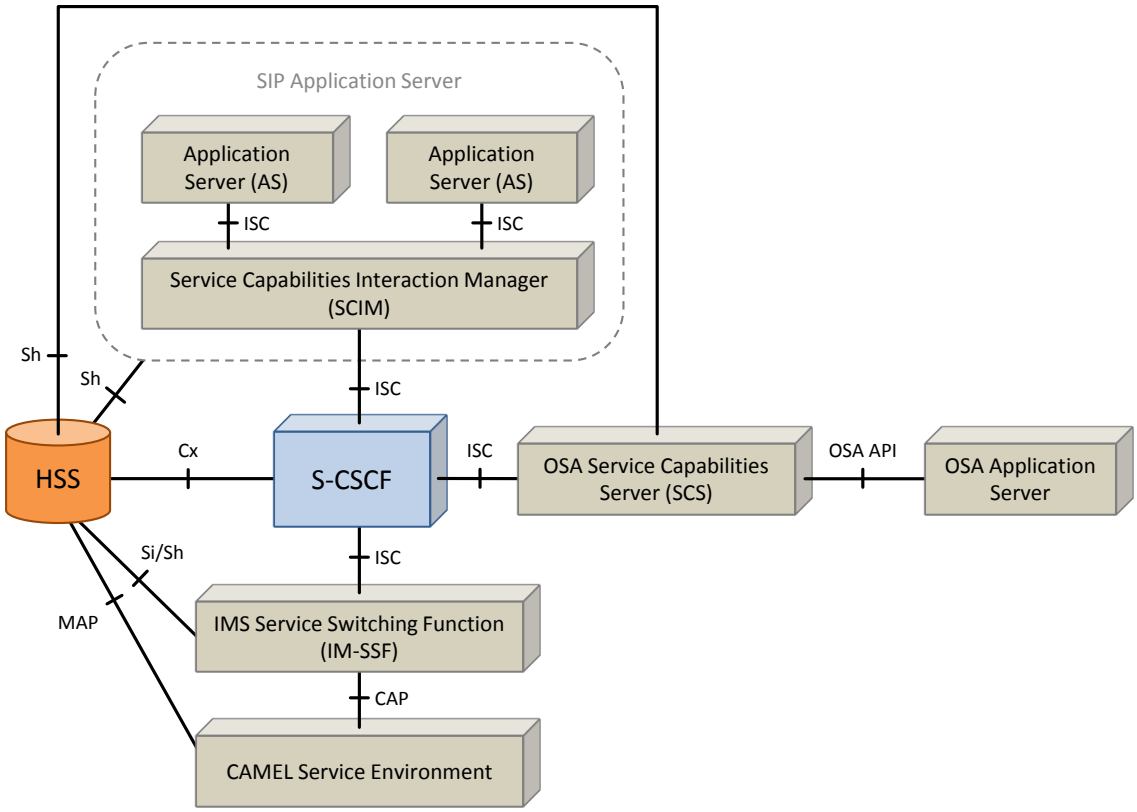

Figure 2.7: Service Capabilities Interaction Manager within the IMS system architecture as specified by 3GPP

tasks for many well known services.

It is also highly important to control where in the network and where along the session path a certain service is included. The respective service interaction control in plain IMS is provisioning by means of iFC setting. The iFC contain a list of services that interact without conflicts if put into the session in that predefined order.

2.1.5 Service Capabilities Interaction Manager

Service routing by means of iFC is relatively inflexible, because it is pre-provisioned. On consequence is that all services needed by a user are always included into the session. Flexible decisions regarding the needed set of services based on, for example, the user's current context are not possible. The need for allowing more dynamic behavior was recognized by 3GPP. As a consequence a new functional node was added to the IMS service layer. The Service Capabilities Interaction Manager was introduced by the technical specification 23.218 [36] in 3GPP release 7. It is a node in the role of a SIP application server and consequently interacts with the CSCF over the ISC interface. The SCIM does not provide services directly, but it acts as service router. A service request from the CSCF

is dynamically routed to the SIP Application Server, where the wanted service is offered. The interface between the SCIM and the SIP application servers is again the ISC interface.

The architecture of an IMS system using a SCIM is shown in Figure 2.7 as specified in the 3GPP TS 23.218 [36]. While this technical specification introduces the SCIM concept and roughly outlines its role, further details of its implementation and inter-work are not standardized. This leaves a lot of room for interpretation and for innovative solutions by IMS vendors and integrators.

Figure 2.7 shows also how CAMEL and Open Services Access (OSA) are connected to IMS. Legacy CAMEL services are integrated by means of an IMS specific Service Switching Function (IM-SSF). Open Services Access provides a Parley/OSA API as specified by ETSI and 3GPP [37]. Parley/OSA provides access an abstraction of network control functions through a web services API. Using this API a 3rd party can, for example, send SMS, initiate and control calls or request the terminal location.

The Ericsson Composition Engine is one possible implementation of a SCIM. It can do flexible application routing. The Ericsson Composition Engine was developed independently of the standardization of a SCIM based on the same root problem of insufficient flexibility of iFC based service routing in IMS. Nevertheless, it can well cover the SCIM role. However, the Ericsson Composition Engine implements service handling and orchestration techniques that by far exceeds the scope of simple application routing. It will be introduced in more detail in Chapter 2.2

2.1.6 SIP Application Routing with SIP Servlets on SIP Enabled Java Applications Servers

SIP servlets are the base for enabling flexible application routing. They are platform independent Java server side application components for SIP signaling. They implement a SIP protocol stack creating a SIP enabled JEE Application Server. SIP servlets are build based on the generic servlet API provided by the Java Servlet Specification [38].

Servlets are managed by servlet containers, which are application server extensions that provide the servlet API and handle the full life-cycle of servlets. SIP servlets are respectively managed by a SIP servlet container. The container also routes application requests to the respective applications that are implemented as servlets. Therefore, it decides which applications need to be invoked and in which order. There are two Java Standardization Requests (JSR) that specify a SIP specific servlet API. JSR-116 [39] and JSR-289 [40]. JEE application servers with SIP servlet containers according to JSR-116 and JSR-289 are, for example Glassfish, IBM WebSphere Application Server or Oracle WebLogic.

In addition to the ability inherited from the servlet API of responding to incoming requests, the SIP Servlet API as specified in JSR-116 and JSR-289 supports a number of important capabilities:

- It allows to generate multiple responses.

- It also allows servlets to act as proxies for requests. Thus, servlets can forward requests possibly to multiple destinations.

- Servlets can initiate requests themselves rather than only replying. Consequently, SIP servlets may receive responses as well as requests.

In addition to that, the event model of the SIP servlet API is asynchronous. This means that SIP applications are not obliged to respond directly to a request coming from the container. They may rather initiate some other action, return the control to the container and respond to the request at a later point in time. This process being asynchronous simplifies the programming of event driven services and allows an application such as a back to back user agent not to hog threads while waiting for a potentially long-lived transaction to complete.

Another important property is that application routing for initial requests and subsequent requests are propagated along the same application path. An initial request is a request for which the container has no previous routing knowledge. Each application has a set of rules associated with it that determine the routing of initial requests. These rules specify under which circumstances, i.e. for which requests, the application is interested in executing. Application rules are specified declaratively and are carried in the deployment descriptor of an application. The container matches initial requests against the set of rules and invokes applications respectively. Responses always follow the reverse of the path taken by the corresponding request.

If an initial request results in a SIP dialog being established and at least one application is on the SIP signaling path the container maintains this path state so that other requests in the same dialog can be routed to the same set of applications. Requests that are routed based on such a path state are called subsequent requests. Unlike initial requests, a subsequent request is not directly dispatched to applications based on pre-configured rules. The signaling is rather propagated along the path of the corresponding initial request of the session in forward or reverse direction.

When passed to applications, SIP servlet requests and responses are always associated with a SIP session. A combination of an application, the application session, and a corresponding external SIP session is called a message context. When routing subsequent requests, containers must determine which applications to invoke and also the context in which to invoke them. This routing scheme of subsequent requests and responses along an established application path clearly reflects the call or session path typical in telecommunication.

The SIP servlet specification JSR-116 standardizes the functionality of SIP containers. It defines a set of SIP servlet API providing a framework for SIP services implemented as servlets. JSR-116 formalizes the method of how such servlets access SIP functionality, react to incoming/outgoing SIP messages and interact with the SIP container. For application deployment, the specification introduces the deployment descriptor format. It also defines some rules that specify when and in which order SIP applications shall be invoked. It is allowed and possible that the same SIP Container has several applications and servlets

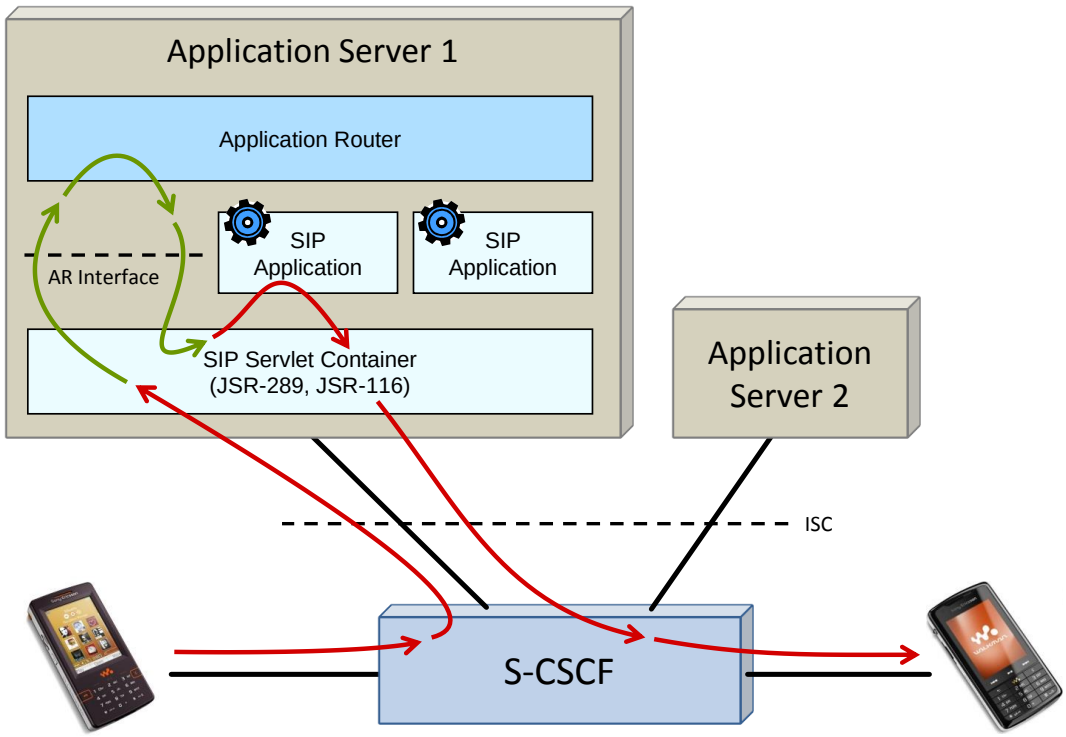

Figure 2.8: Application routing within a SIP application server based on a SIP servlet container

that can be invoked on the same incoming SIP packet, according to the rules described in their deployment descriptor. Using these features a basic composition functionality is realized. It allows for static composition of linear sequential chains of SIP servlets.

JSR 116 also describes the behavior of a SIP container at run time. It specifies how SIP requests are propagated by the container towards the applications and servlets that are responsible for their processing. It also specifies the management of run time dependencies between the SIP applications and servlets belonging to them.

JSR-289 extends the SIP servlets specification taking into account experiences gained with JSR-116. One of the added features is an improved support for composition of SIP services through the introduction of an interface for a dedicated application router. It furthermore improves the possibility to interact with other technologies, for example Enterprise Java Beans (EJB), Web Services or HTTP Servlets deployed on the same application server. Figure 2.8 shows how IMS application routing can be done within a SIP servlet container based application server.

SIP servlets with the capabilities of JSR-116 and JSR-289 constitute the base on which the Ericsson Composition Engine introduces a flexible composition mechanism that can take the role of a SCIM in the IMS context. More specifically the Ericsson Composition Engine takes the role os an application router.

2.2 Ericsson Composition Engine

The rule based application routing capabilities of SIP servlet containers as described in Chapter 2.1.6 allow static composition of applications, with constituent application components build as servlets. Based on this technology, application servers with a SIP servlet container can already take the role of a SCIM. Next to the iFC based service routing in the CSCF, the invocation rules in the servlet container provide a second level of SIP service routing with full control of persistent session paths through participating applications. However, the invocation rules are applied at application deployment, and thus, they are also relatively static.

The Ericsson Composition Engine was originally developed in order to bring considerably more flexibility to SIP service handling through dynamic and data driven service composition. The expectation was that time to market could be considerably reduced when a powerful composition methodology is available. Especially the higher abstraction in implementation together with an increased re-use potential application components implemented with clear functional separation and well defined API would enable this. It follows in this respect the methodologies of Service Oriented Architectures trying to gain similar advantages.

The following chapters describe the Ericsson Composition Engine. It is the base composition technology used in this thesis and it therefore gets a detailed introduction. It contains a few unique properties of a composition technology that are particularly interesting in the context of aspect oriented software development and management. One example would be constraint based service selection at run time. Furthermore, the Ericsson Composition Engine fully supports the particular service usage and service interaction schemes of telecommunication service sessions. This is in particular relevant when the goal is to investigate an AOP mechanism in the context of telecommunication service development.

Please note that there is also an Ericsson Product that is referred to as Ericsson Composition Engine. It has inherited the name from the Ericsson Composition Engine that was originally developed as research prototype. The product contains the functionality of the research prototype extended to a product-grade implementation, but it also contains many more components, for example a full JEE application server. Within the scope of this thesis the term Ericsson Composition Engine only refers to the composition execution engine of the research prototype, and thus, only to the composition mechanism on top of application server and servlet containers. The conceptual base of this Ericsson Composition Engine was introduced by [41].

2.2.1 Service Composition for Telecommunication Services

When creating a service composition technology it is essential to first understand the unique characteristics of the targeted service domain. Business process execution based on the Business Process Execution Language (BPEL) [6] or the Business Process Modeling Notation (BPMN) [7] are well established and widely known techniques for service composition. For both services are usually considered to be stateless. They are invoked and

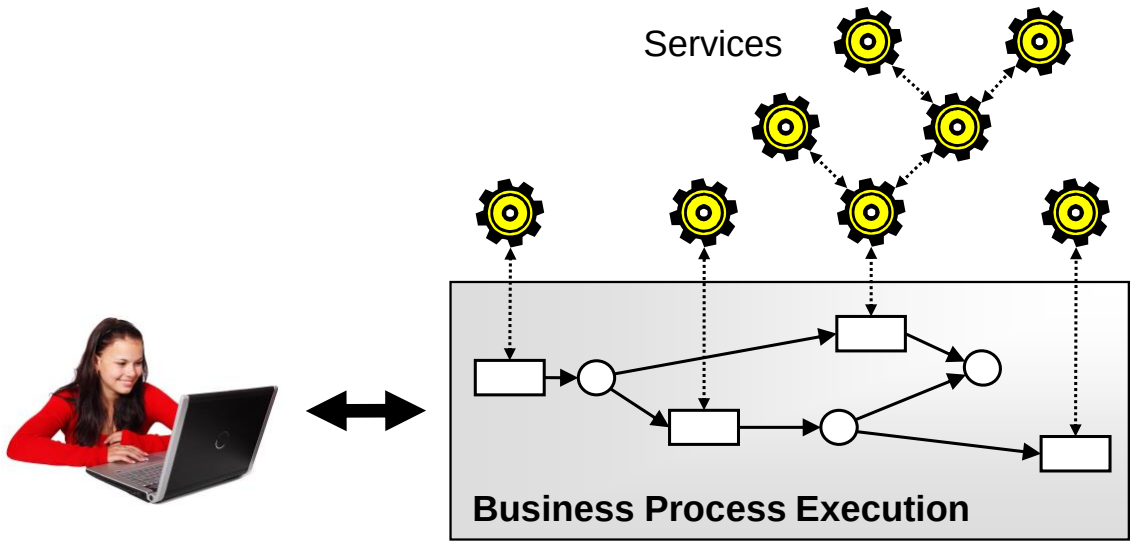

Figure 2.9: Execution of business process orchestrations

they communicate through a request response dialog. SOA/Web Services is an example of a service technology that is often used together with BPEL or BPMN.

The composition in BPEL or BPMN has a work flow character and the work flow is executed by a business process execution engine. The composition execution is started on user request and the business process execution engine orchestrates the needed services by instantiating them according to work flow specific rules.

An invoked service instance performs the required service and replies with the respective result. The service instance is usually stateless, not persistent and synchronous. If one constituent service has finished, the next service is executed based on the composition rules expressed in BPEL or BPMN. A service instance can of course invoke other services in order to process the result. Thus, the service instances being orchestrated might show a tree-like structure as shown in Figure 2.9. This service tree is dynamically build and not persistent. Only the session of the business process execution itself is persistent, giving the business process execution engine a central controlling role.

As shown in previous chapters, telecommunication services are usually persistent within the context of an end-to-end user session. They are also logically allocated along a session or call path between two endpoints. Once a service is persistently included in the session path, it subscribes to and observes the signaling passing by. The service decides itself based on its state and the observed signaling when it needs to get active. The resulting action can be a change of existing signaling, termination of requests with generation of replies or the sending of new requests. No central coordinator or composer is needed that controls how the services operate and how they apply their application business logic. The services interact collaboratively with each other rather than being controlled by a central entity. This of course demands a high degree of service interaction control and detailed rules what a single service is allowed to do with respect to the session state. This is reached through

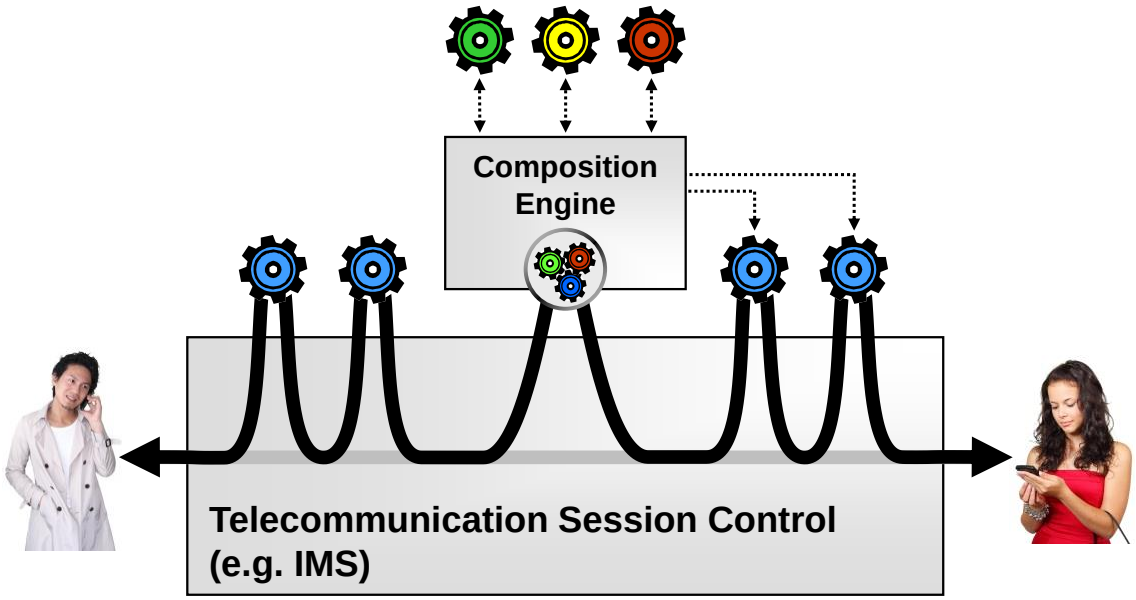

Figure 2.10: Execution of heterogeneous service composition

detailed standardization.

Although telecommunication services apply their function without central control, there is still room for composition. However, the composition task is defined differently than for business process execution. The composer needs to determine which services are needed within a session and where in the service chain a particular service needs to be allocated. Once the service is in the chain, the task of the composer is first of all over. The actual function is provided only by interaction of the persistent services with each other and with the underlying IMS system without the composer being involved.

The composer might however be deployed again when changes to the service chain setup are needed. In order to notice the need for changing in the service chain setup, the composition execution engine might itself listen to the control protocol at defined locations on the session path. The composer instance becomes itself a persistent part of the session path, similar to services. Nevertheless, outside the chain setup and later alterations, the composition engine stays passive. This difference in topology and roles within the service usage compared to BPEL and BPMN is one of the main reasons, why the Ericsson Composition Engine was developed rather than using the already existing and widely known business process modeling and execution solution.

The Ericsson Composition Engine and SIP services on the end-to-end service chain are shown in Figure 2.10. Please note that the composer itself is logically also allocated on the service chain, but it mainly determines which other services shall be added. Please also note, that the Ericsson Composition Engine can also act the way other business process execution engines do and invoke, for example web services. Their reply can then, for example, be used in the setup decisions for the telecommunication session. Thus, the Ericsson Composition Engine natively supports both service usage models combined. It

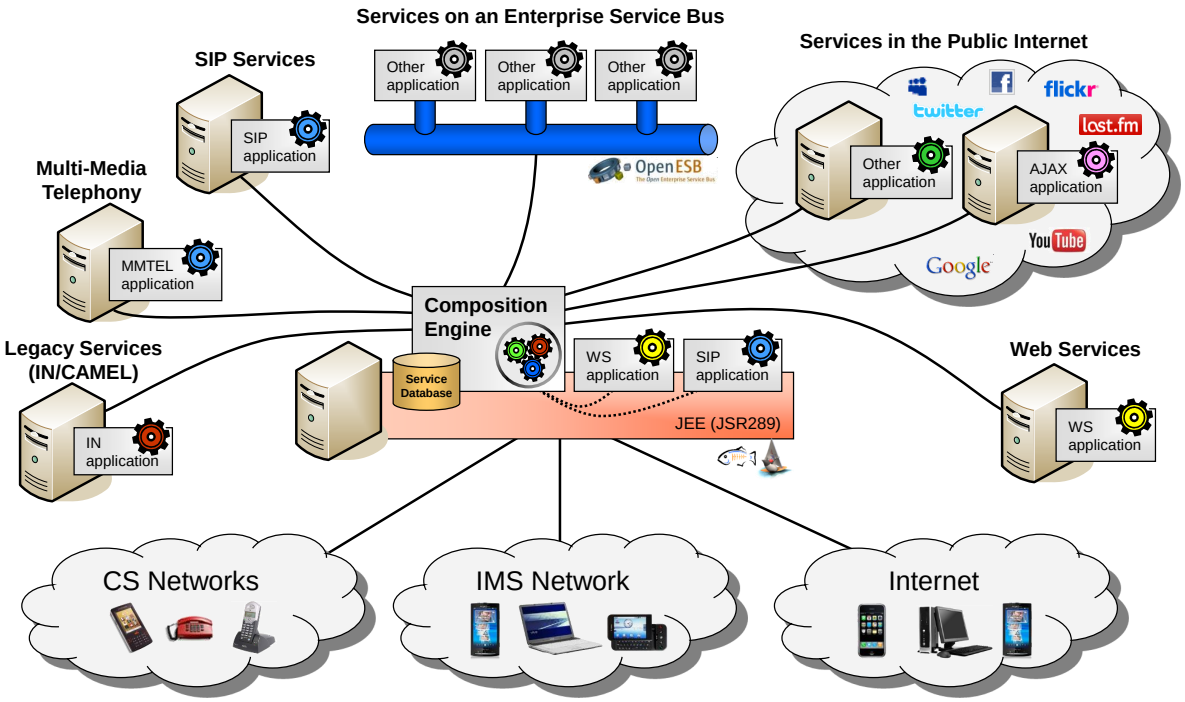

Figure 2.11: Overview of the Ericsson Composition Engine within the supported service technologies

is therefore able to natively utilize SOA type services within a telecommunication context and vice versa.

This native support of multiple service technologies is fundamentally different from older approaches to integrate telecommunication and SOA services. Parley/OSA, for example, separates the domains through a predefined web-service interface. SOA type services can only reach those aspects of a telecommunication session that are explicitly exposed through the Parley/OSA API. New functionality would usually demand a change of this API. This is the reason why the use of Parley/OSA is practically limited to applications where only predefined standard telecommunication services are involved. With the Ericsson Composition Engine an application is natively composed from telecommunication services and SOA services together without separating them. The SOA services can directly be involved based on all the details of IMS/SIP session control. This allows a much higher degree of flexibility for developing new converged applications.

Figure 2.11 provides an overview of all the service technologies natively supported by the Ericsson Composition Engine. From the telecommunication domain there are SIP Services and the integration with legacy IN/CAMEL services. From the IT domain, there is support for Web Services (WS), Representational State Transfer (REST) [42] services and Asynchronous JavaScript and XML (AJAX) [43] Services. Furthermore, the Ericsson Composition Engine can be integrated with an enterprise service bus (ESB), reaching all services deployed on the bus for composition. This support for multiple technologies

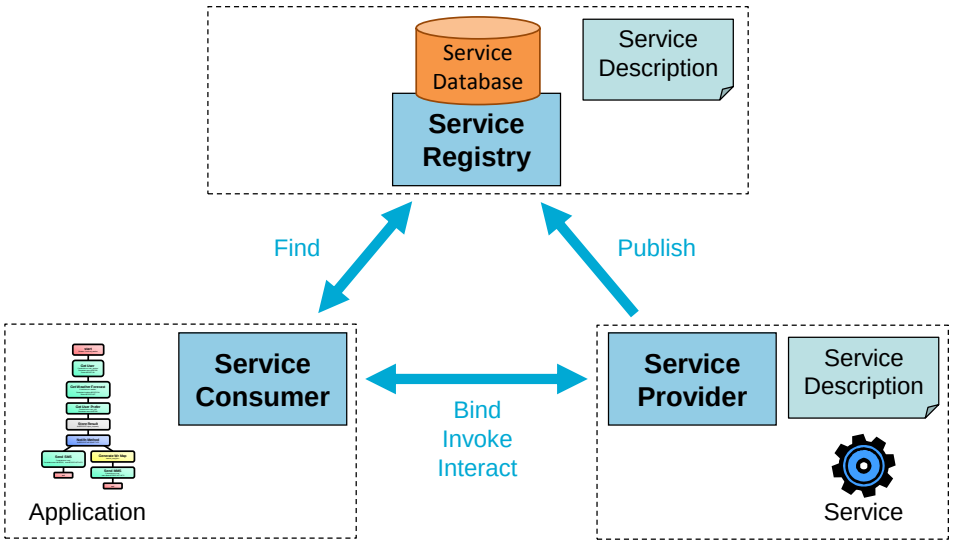

Figure 2.12: Publish-find-bind scheme for flexible service binding at run time

includes that the composite application itself can also be exposed based on any of these service technologies. It can therefore be invoked from legacy telecommunication networks, from SIP based IMS controlled next generation networks or from IP based Internet.

2.2.2 Service oriented Architecture for telecommunication services in IMS

A Service Oriented Architecture (SOA) is based on a set of general concepts for developing and handling of applications. This chapter provides an overview of the essential concepts [5] with comments on how they are transferred into the telecommunication domain by the Ericsson Composition Engine. The following chapters provide more detail on the Ericsson Composition Engine being developed following these SOA principles, thus, building a SOA for telecommunication.

Loose Coupling
Loose coupling represents a relationship between a service and an application that integrates that service. It allows the underlying logic of the service to change with minimal or no impact on the other services utilized within the same SOA. Loose coupling is a key principle of service orientation and a key property for separation of concerns. Implementing services as loosely coupled pieces of software is an essential prerequisite for most of the other key principles of service orientation.

Service Contract
Service contracts represent service descriptions and other binding information, describing how a service can be accessed programmatically. Often a service repository is used. It is a database of available service contracts. New services publish their contracts and

binding information through this repository allowing applications to dynamically find and ultimately bind instances of the needed services at run time. In the context of introducing SOA for IMS services this publish-find-bind scheme that is shown in Figure 2.12 was kept and extended by the introduction of service descriptions for SIP services.

Abstraction
Abstraction of underlying logic means that a service publicly exposes only logic described in the service contract, hiding the implementation details. This means that services interact with each other only via their published interfaces. This concept was kept also for descriptions of SIP services.

Autonomy
Autonomy means that a service only controls the logic it encapsulate. This concept refers to the principle of dividing application logic into a set of autonomous services. It is essential for achieving loose coupling, reusability, and composability. Also telecommunication services follow this principle.

Reusability
Reusability is achieved by distributing application logic among services in such a way that each service can potentially be used by more than one application. In this respect it is preferable to implement each functional concern in a separate service rather than implementing more complex services that fulfill multiple purposes.

Composability
Composability represents the ability of services to be grouped into composite service applications. The composer coordinate the service invocation and the exchange of data between services. The Ericsson Composition Engine with its unique composition language is yet another example of a composer that is able to compose in a similar way as BPEL and BPMN based engines. However, SIP services behave considerably different as explained in Chapter 2.2.1. Composability is still reached, but it practically means a different relation between the service and the composer.

Statelessness
Statelessness means that services don't maintain their state specific to an activity and across invocations. In the traditional SOA sense, building stateless services encourage loose coupling, reusability, and composability because state often keeps knowledge about context and other services being used. Statelessness requires a request can be answered by a service without considering other requests and the responses given. Thus, all kinds of session management and inter-service coordination is entirely the task of the composer. In telecommunication service environments this is not the case due to the role of services within a service chain. Subsequent requests within a session context based on stateful protocols, such as SIP, which propagate messages along an established session path are used. Thus, a SOA approach for telecommunication needs to consider stateful services and multiple asynchronous requests and responses per service instance. This does not mean,

that every composed service is required to support state. For existing SOA services there is no change and the composer takes care of the coordination. Nevertheless, the composer is also enabled to interact with stateful services if necessary. Thus, depending on the type of a selected service, the composer will interact differently with it.

Interoperability
Interoperability between services is achieved as long as the services interact with each other through interfaces that are platform- and implementation-independent. In practice this is easy to achieve within a single family of service technologies, for example, if only Web Services or only SIP services are used. In these cases the communication between the services is well defined. If compositions shall consist of services from different technological families, a mediator role is needed that helps bridging between different protocols and formats. A composer, such as the Ericsson Composition Engine, is able to take this mediator role as explained in the following chapters.

Discoverablity
Discoverability refers to the availability of a standard mechanisms that enable service descriptions to be discovered by service users/consumers. Universal Description, Discovery, and Integration (UDDI) [44] specification provides such a mechanism, which allows for publishing service descriptions documents in an XML-based registry, thus, making them available for public use.

The SOA concepts as such are not tied to a concrete technology or family of service technologies. Nevertheless, there exists a set of technologies and related protocols that allow to implement applications according to SOA principles. The most prominent example is Web Services and related protocols and description languages. UDDI [44] is used to publish and find services through a service registry. The Web Services Description Language [45] is used for describing the services as part of the service contract. The Simple Object Access Protocol (SOAP) [46] is used for invocation of services and the communication between the service and the application.

Not all of the SOA concepts can be applied to the telecommunication domain as originally defined. The most significant diversion is the need for stateful services with asynchronous interaction. This difference was also one of the drivers for starting to develop a new and unique telecommunication specific service composition technology.

2.2.3 Components of the Ericsson Composition Engine

The term Ericsson Composition Engine only refers to the actual composition execution engine. It is accompanied by an environment of other components that together enable design and execution of composite services:

Service Database
A service repository that allows to publish service descriptions and find the services to be integrated into compositions. The Ericsson Composition Engine uses the service database to find and integrate constituent services at run time.

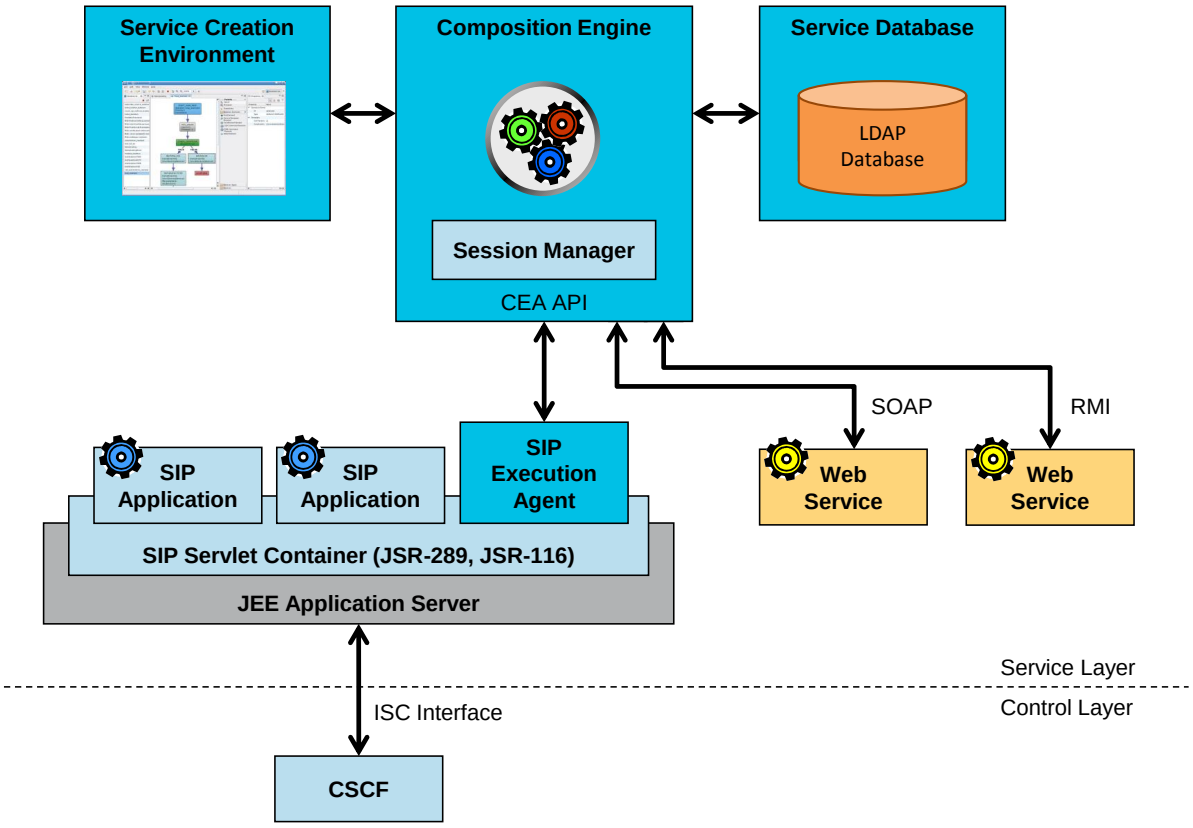

Figure 2.13: Components of the composition environment and integration with IMS, SIP and other services

Service Creation Environment

The service creation environment (SCE) is an integrated development environment for composite service applications. It contains a composition editor, a management front-end to the service database and debugging capabilities for composition sessions. Through the database front-end it allows the import and editing of service descriptions.

Ericsson Composition Engine

The Ericsson Composition Engine is the run time environment for service compositions. It performs service selection, initiates the invocation of services and controls the composition session and execution flow.

Composition Execution Agent

A composition execution agent (CEA) acts on the logic of the composition engine with respect to a particular service technology. The SIP execution agent, for example, interacts with IMS and other SIP applications through a SIP servlet container.

The main components of the development and execution environment around the Ericsson Composition Engine are shown in Figure 2.13. This figure also shows its deployment on

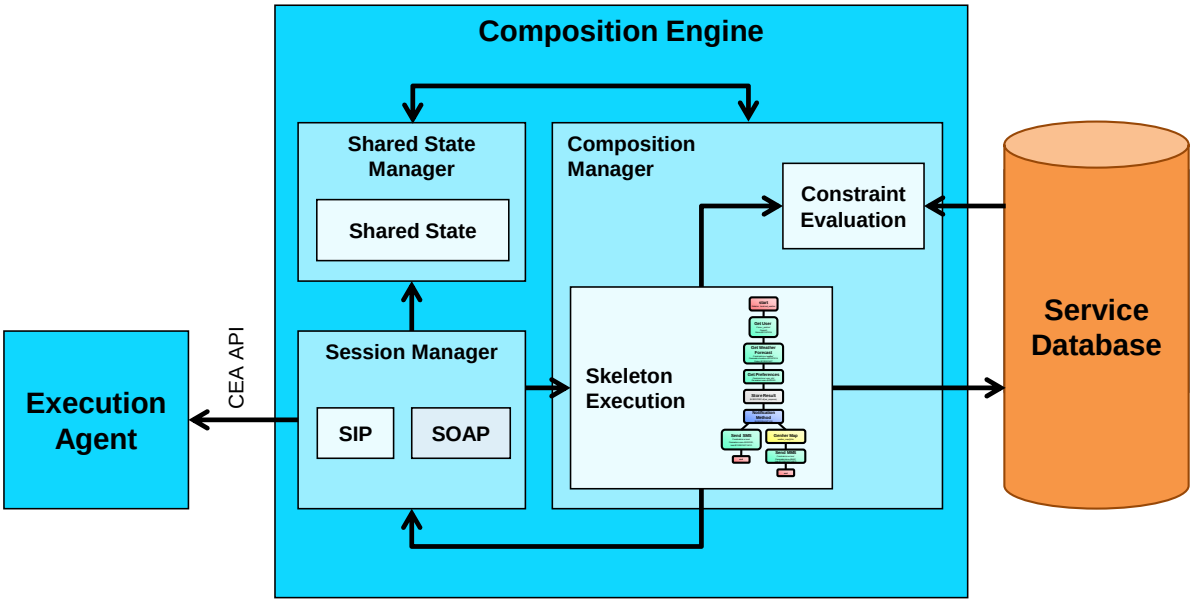

Figure 2.14: The internal structure of the Ericsson Composition Engine

a JEE application server and the relation to other services and applications. The composition engine itself is deployed on an Enterprise Java Beans Container. The SIP execution agent integrates into the SIP environment provided by a SIP servlet container. Thus, the SIP CEA is a SIP servlet.

The basic idea of composition execution agents is that the service technology and protocol related part of a constituent service is separated from the composition logic and service selection. By adding further execution agents it is possible to utilize further service technologies without changing the composition core. For using Web Services and other EJB based applications no dedicated execution agent is necessary. The Composition Engine core communicates with the execution agent through a dedicated API.

All components of the Ericsson Composition Engine together establish a complete service composition framework that covers the full life-cycle of composite applications including development, test and execution.

The internal structure of the Composition Engine consists of three main functions. The Composition Manager executes the composite service by interpreting a composite application skeleton that contains all composition rules. A central part of this execution is the selection of constituent services through constraint evaluation against the service database [47]. The shared state manager contains the state, and therefore, all common session run time data for the execution of this instance of a composite application. The third component is the session manager. It coordinates the execution sessions for constituent services that were selected by the composition manager. The actual execution of constituent services is performed by a composition execution agent that is directly interact-

ing with the session manager. Figure 2.14 shows the core component and their interfaces.

2.2.4 Service Descriptions for SIP and Other Services

When creating a SOA based service environment for SIP services, the introduction of a publish-find-bind scheme is essential. It is the base for reaching most of the SOA characteristics and paves the way for service composition. For the Ericsson Composition Engine an LDAP database serves as service repository. It uses a database schema that allows specifying abstract service descriptions together with binding information. The concept was generalized in order to get a service database that not only act as registry and repository for SIP services, but also for other service technologies. For example, WSDL based service descriptions can be used directly for publishing Web Services.

The LDAP service description schema allows the use of abstract service descriptions. Any number of additional parameters can be added to a service description. They are used, for example, to specify the abstract function of the service or contextual information, for example the identity of the service provider or the version of the service. These descriptive parameters are independent of the service technology.

The binding information part of a service description depends on the service technology. For SIP services it specifies the information that is needed by a JSR-116 or JSR-289 SIP servlet container for invoking a SIP application. For web services the binding information is taken directly from WSDL import.

The service description used with the Ericsson Composition Engine is therefore quite similar to other service descriptions within a SOA. The main differences are the possibility to have SIP binding information and the extended abstract descriptions. This abstract description of the service is to a great extend independent of the technology used for implementing, hosting and publishing the service. In the Ericsson Composition Engine, this abstract and technology independent part of the service description is the base for selecting constituent services at run time. Consequently the selection of a service becomes entirely independently of the technological and implementation details of the service. The binding information is not considered by the composition manager when selecting a service. It is only used when in a second step the service execution is invoked.

2.2.5 The Skeleton and the Composition Language

The composition language of the Ericsson Composition Engine is based on a relatively simple graphical notation for defining the control logic of the composite application and the selection rules for constituent services. A so called 'skeleton' is used to specify the composition.

The skeleton defines a work flow with the notation convention to start on the top and execute downwards. A start element marks the execution start of the composite application that is described by the skeleton. The composition engine then executes the skeleton downwards until an end element is reached. Six basic elements are available to express the execution semantics of the composition. The execution is synchronous in the

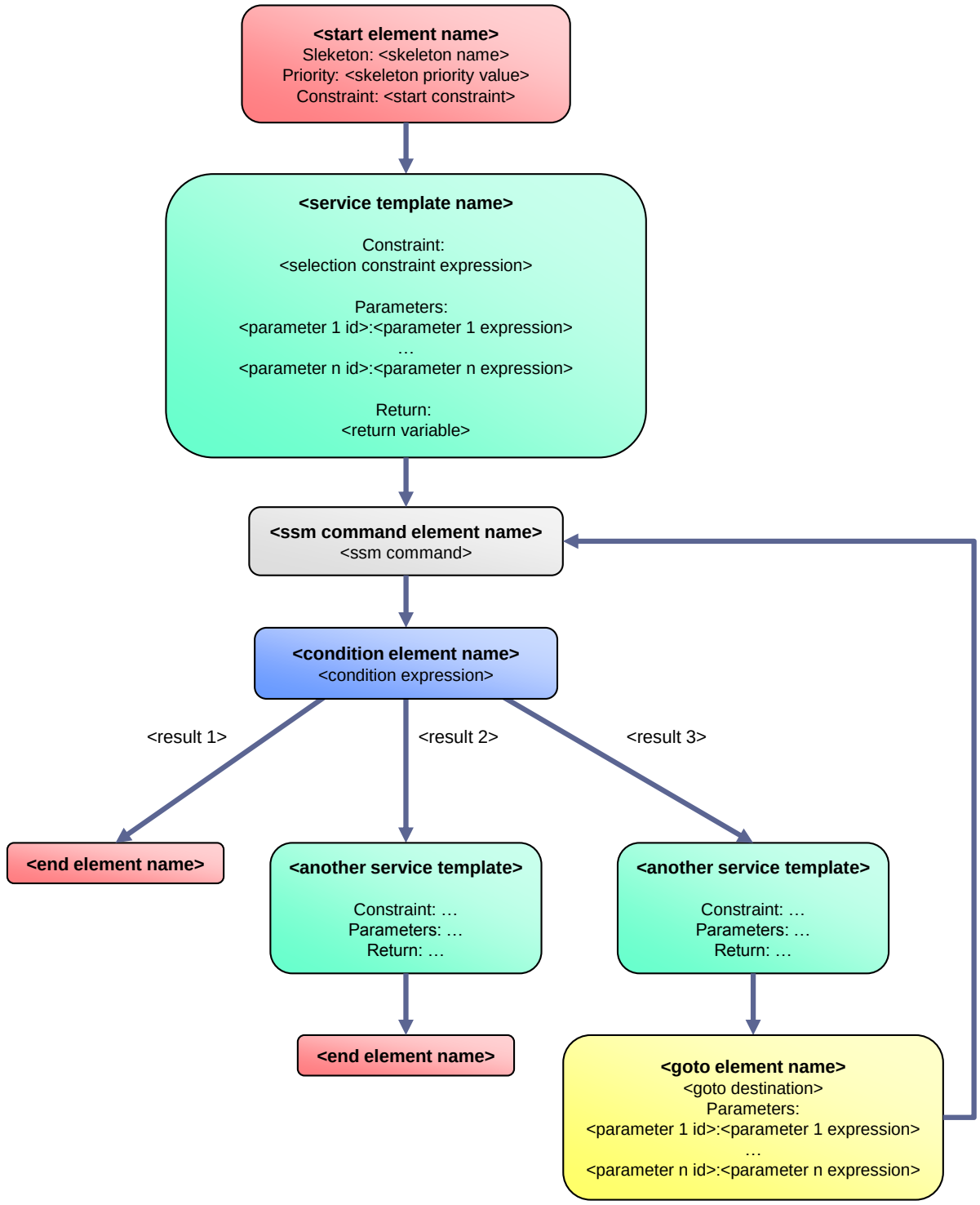

Figure 2.15: A generic skeleton with all skeleton elements

sense, that each element needs to finish until the execution engine proceeds to the next element. It is doing so by following the connecting arrows between the skeleton elements.

Figure 2.15 shows an example skeleton with all available generalized skeleton elements. Common to all skeleton elements is that they have an identifier or name that can be freely chosen. This name appears as the top line in each element.

Start Element
A composite service execution is always started at the start element of a skeleton. This element contains the unique identifier of the skeleton. It contains the execution priority to be used when several skeletons are about to start for a session. The skeletons with higher priority are executed first and only one skeleton at a time is executed for a parent session. A skeleton constraint allows to specify conditions for the skeleton to start. This is frequently used, for example, to filter out only those SIP requests the skeleton needs to be executed for.

End Element
The end element terminates the composition session once one element of this type is reached. This only terminates the composition, it does not terminate the parent SIP session. The end element does not have parameters.

Service Template
The service template is a placeholder for a constituent service within the composition. The service template needs to be instantiated by a service. Executing the service template first of all means selecting this constituent service by fulfilling a selection constraint specified by the constraints parameter. Based on this constraint a request towards the service database is performed in order to find available services that fulfill all functional and non-functional requirements to be executed. Based on the related binding information a matching service is invoked and executed. When the service answers with a reply, the service template execution is finished and the skeleton execution proceeds. The service template also allows to specify the invocation and return parameters the constituent service.

SSM Command
The shared state manager (SSM) command allows changes to the shared state. This can be, for example, the assignment of a value to a shared state variable. The command expression can contain basic numeric calculations, strings and other variables.

Condition Element
The condition element is a structural branching element. The execution branch that matches the condition is selected.

Goto Element
The goto element allows to divert the execution flow either to another skeleton, or to proceed execution at another location within the same skeleton. In order to specify the destination within the same skeleton the unique IDs of skeleton elements are used. The

execution continues with that skeleton element. If the destination points to another skeleton, the execution is continued by starting that skeleton. The destination skeleton inherits the shared state, thus, it can proceed execution with all run time data. Additionally, invocation parameters can be specified in the Goto Element. They will be available in the shared state of the destination skeleton. Once the invoked skeleton finishes, the execution proceeds in the calling skeleton after the goto element.

2.2.6 Constraint-Based Service Selection and Invocation Through Execution Agents

The Ericsson Composition Engine dynamically selects and binds constituent services in order to instantiate service templates. The entire selection of constituent services and not only the binding is executed at run time. It is based on abstract constraints defined on service templates. The composition developer specifies the constraint at deign time. It is an expression of all requirements a suitable constituent service needs to fulfill. It first of all describes the wanted function, but the mechanism can also be used for requesting context of the service candidate, for example its providing organization. Furthermore, non-functional properties of the service can be demanded in the selection process.

When executing a service template the composition engine builds an LDAP query towards the service database from the constraints expression. The result of this query is a list of all registered services that fulfill the constraint. Each of these services is suitable to be invoked. This selection mechanism only considers the abstract part of the service descriptions. Therefore, it is entirely agnostic of the service technology that will is chosen. The service database can propose, for example, a mixed list of Web services, AJAX services and SIP services as suitable candidates for instantiating a service template.

This service selection mechanism means that the Ericsson Composition Engine decides ad-hoc at run time which constituent service is chosen and used and what technology and protocol needs to be used in order to execute it. This also means that each execution of the composite application might lead to a completely different set of services being used to instantiate the composition depending what is found in the service database at the particular time of execution. This is highly flexible and adaptive to various service environments. The composition execution by interpreting a skeleton and selecting services based on a service database is shown in Figure 2.16

The Composition engine selects one of the services proposed by the service database. The following action is asking the composition session manager to initiate the execution of this service. The composition manager decides, based on the binding information in the service description, which execution agent is suitable for executing this particular service. Thus, the selection is service technology agnostic, but the execution is performed within a suitable environment provided by a service technology specific execution agent. If, for example, a SIP service was selected, the SIP execution agent would initiate a respective SIP request using the SIP servlet container according to JSR-289. If the selected service is a web service, a SOAP request is sent respectively.

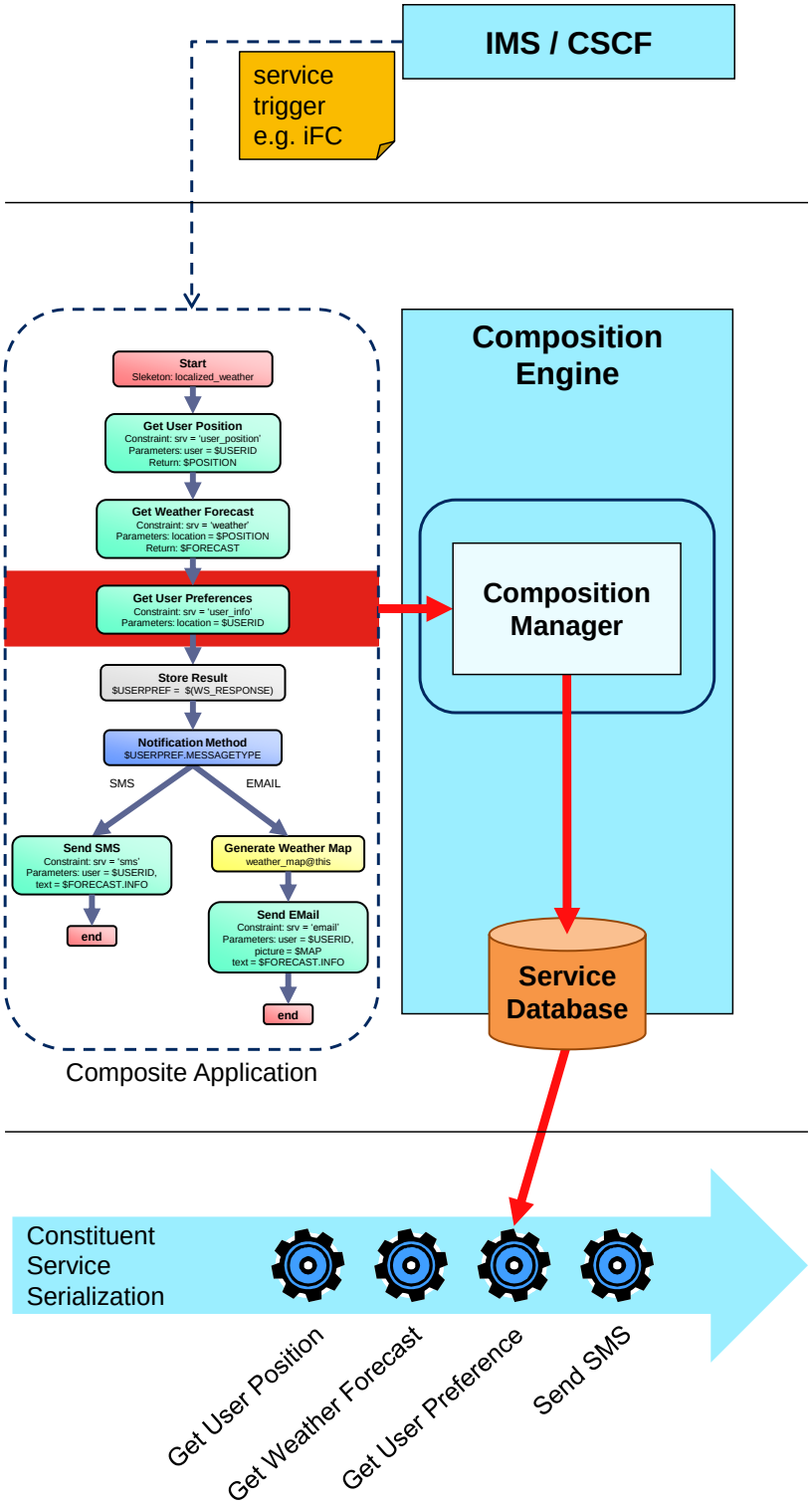

Figure 2.16: Skeleton execution in order to assemble the service execution serialization

Further service technologies can be added by implementing additional Execution Agents. The Skeleton execution and service selection would not be effected this addition, because it is service technology agnostic. The fundamentally different service usage of SIP service compared to, for example, Web services is mainly handled by the execution agent. This task devision between the execution agents and the composition core allows the Ericsson Composition Engine can interact with both types of service usage.

2.2.7 Shared State

Each composition execution session has a common shared state. This refers to a set of run time variables storing all session data. These shared state variables can be used to populate parameters of service invocations or they can be used in branching conditions as being used in condition elements. They can therefore determine the composition execution. The type of a shared state variable is transparently and automatically selected depending on the kind of values assigned to the variable. Basic string or numeric values would be possible, but also indexed lists and hierarchical structures can be defined. Type conversion is applied transparently.

Protocol messages that were used in order to invoke the composition session are stored in the shared state with all message parameters and data. If the composition was, for example, invoked by a SIP INVITE request, all parameters of the INVITE are available in the shared state at the start of the composition execution. Also prior to sending a SIP message, all its parameters are assembled within the shared state.

Shared state variables are common and shared for all used constituent services. In particular all data related to technologically different services is not separated but kept within variables of the same shared state. This is the base for the ability of the composition engine to mediate between different services while using them together within a single composition. This way the results of, for example, a web service execution can directly be used as a parameter in a SIP service request.

Another important use of shared state variables is within constraints. This means that the actual requirements for a service selection might depend dynamically on data, for example the results of previously executed services. This allows, for example, an external database to determine which services shall be used for a user.

2.2.8 Interaction with IMS Through SIP Servlet Containers

Chapter 2.1.6 has described the usage of SIP servlets and servlet containers based on JSR-116 and JSR-289 for developing applications that interact with and control IMS/SIP sessions. The Ericsson Composition Engine fit into this environment through its SIP execution agent. The SIP CEA is in fact a SIP servlet that is interacting with the servlet container through the application router interface. The composition engine behind the SIP execution agent introduces therefore a flexible application routing mechanism on top of the application routing within the SIP servlet container. The added value lies in the

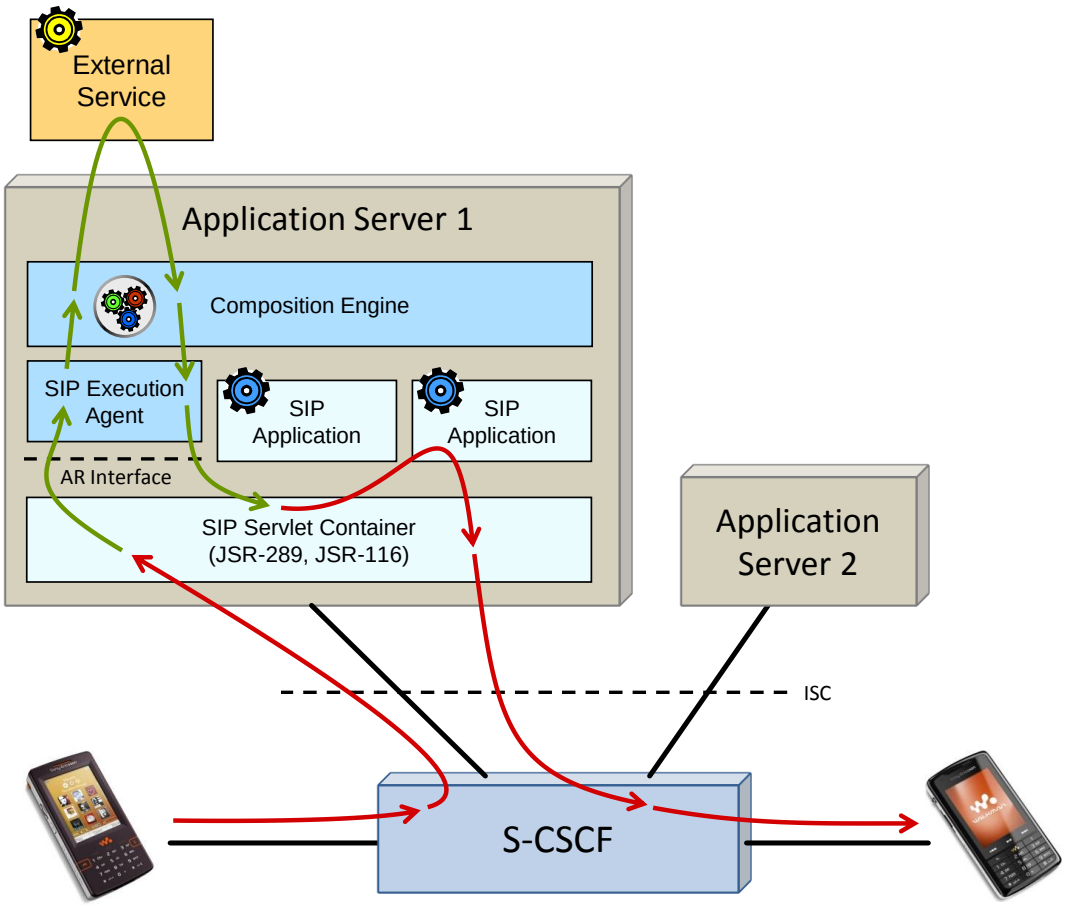

Figure 2.17: The composition engine with CEA as application router in a SIP servlet container

introduction of a service composition methodology for controlling the application routing rather than relying on statically defined routing rules.

Figure 2.17 shows the allocation of the composition engine on the application server. It also shows the path of a SIP request. It is first directed towards the application server by the CSCF based on initial filter criteria. The SIP servlet container then routes the request to the SIP execution agent of the composition engine. This initiates the selection and execution of a composition skeleton. The composite application execution might involve a number of constituent services until one of them is a SIP service. Through the SIP agent and according to JSR-116 or JSR-289 the selected SIP service is initiated with the help of the SIP servlet container.

After the invoked SIP service forwards the SIP request, it is sent on again to the composition engine. The composition execution proceeds and this scheme of service invocations might be repeated for further SIP services. If there are no more SIP services to compose, the composition engine notifies the servlet container. The SIP request is then routed back to the CSCF and towards the session destination. This interaction scheme is defined in

JSR-116 and JSR-289. From SIP servlet container point of view the composition engine acts as inference engine for application routing decisions.

2.2.9 Composition Performance

This thesis puts special emphasis on telecommunication uses cases. This focus implies particular attention to certain timing parameters, such as service response times as experienced by the users at the endpoints of the session path. Especially everything that might delay the signaling in the end-to-end call chain is critical. For the participating service composition engine, the time needed for taking all composition decisions and for requesting the services is the critical parameter. An absolute and generally required threshold for this time can ultimately not be provided. It depends on the operator's network planning but in the proof of concept projects with the Ericsson Composition Engine, operators have asked to not exceed a maximum of 20 ms additional latency introduced by using composition. This figure refers in particular to latency in the initial forward setup of IMS call sessions. It also only refers to the composition task itself and excludes the time needed for message routing and execution of constituent services. This is therefore the time budget granted for gaining the added flexibility provided by the use of composition methodology. The overall forward setup until the destination is reached depends on the number of hops needed due to the number of services and network nodes involved. It can take a few hundred ms up to several seconds.

2.2.10 Context-Aware Composition

Context awareness refers to applications that detect the situation of their usage, reason about it and react flexibly with adapted service behavior. Context is increasingly important due to emerging domains, such as the Internet of things (IoT). Software applications in this domain are potentially exposed to a great variety of situations. Each of them constitutes a context that might demand a specifically adapted behavior from the application. Context awareness is also a central goal on the path towards pervasive computing [18].

Traditional software design methodologies address context by means of static hard-coded rules, which identify a context and apply a variant of the build-in application logic. This logic was created particularly for a specific set of contextual situations. This approach to treating context was also proposed for service composition with the introduction of Event-Condition-Action (ECA) rules in the composition design patterns. In this respect the event refers to a change in the data that potentially indicates a context change. This initiates an evaluation of the condition. It concludes the current context and consequently initiates the related action as direct contextual response. In the composition process these rules are used to determine, which constituent services needs to be executed [48, 49].

Using the Event-Condition-Action design pattern for composite applications introduces a structured treatment of context by means of a rules-based control layer. Nevertheless, it still establishes a tight coupling between criteria, which identify the contextual situation, and the reaction of the application by means of executed functions or services. In other

words, it introduces verity, but not flexible adaptation to contexts, which were not explicitly considered at design-time.

A more flexible treatment of context can be reached by introducing abstract semantic models of the environment [50]. This model allows to define flexible rules for context identification and for selection of a suitable service response. The flexibility can be reached by decoupling rules and their semantic from the implementation of services and composite applications. Given that also the possible functions of the application are described semantically, the identification of the context and the matching between a context and a service response follow the knowledge contained in the model. Adaptation to new context would be flexibly done through changes in the model.

2.2.11 Composition Engine Capabilities Comparison

The Ericsson Composition Engine contains a couple of capabilities that are not available from widely used and well understood composition technologies based on BPEL and BPMN. The following are the main differences and similarities:

Integration into the Service Path:

Telecommunication services are components within the end-to-end service path. They participate in the session signaling and they are aware of the session state and typically maintain an own internal state machine as explained in Chapter 2.2.1. They are in particular not operating using a stateless request-response scheme, which is typical for service usage in SOA-type compositions based on BPEL or BPMN. The role of a composition engine for telecommunication service sessions is orchestrating otherwise self-contained services into a session path. After this initial placement, the services are self-contained rather than further controlled by the composer. The latter would be the typical service usage scheme of BPEL and BPMN based engines. This is also reflected in the different signaling schema of SOAP versus SIP. The service invocation scheme of BPEL and BPMN engines is conceptually incompatible to SIP services. The differences are fundamental to a degree that designing a new composition engine and related language rather than trying to adapt existing ones became the most efficient solution.

Support of Heterogeneous Services:

Over decades there were many service technologies developed resulting in a diverse base of services in operation. SIP based services are just the latest development following, for example, IN/CAMEL services. This installed base makes a great investment that ideally needs to be preserved with the introduction of new technologies. Therefore, it is essential for the acceptance of a composition technology to be aware of multiple existing service technologies and to be extensible with respect to new future developments. This requirement resulted in the architectural decision to separate technology-aware composition

execution agents from technology-agnostic composition logic. This architectural separation is fundamental and not available in typical BPEL and BPMN based engines.

Constraint Based Service Seletcion:

This capability introduces run time selection of constituent services based on selection rules, which are expressed by constraints. This capability is not strictly necessary in order to meet domain specific requirements of telecommunication services. However, binding information in service descriptions, as typically used in SOA, is specific to web services. The technology-agnostic composition core lifts the composition logic, and with it the related service descriptions, on a higher and technology independent abstraction level. The specification of which services are to be selected became naturally a rules-like character. Exposing these rules to run time modification was a relatively small addition for added flexibility and expressiveness.

Work Flow Nature of Compositions:

In this respect the Ericsson Composition Engine is similar to BPEL or BPMN engines. The composition execution and the service serialization is determined by interpreting control elements arranged in a work flow pattern. The major difference is that the graphical language of the Ericsson Composition Engine defines only the most basic work flow control elements. There is, for example, no explicit semantic for coordinating parallel processes or for data synchronization. This does not make the Ericsson composition Engine the preferred choice for execution of complex business processes.

2.2.12 Composite Service Examples

The following examples demonstrate the use of the Ericsson Composition Engine in typical use cases:

Example: Automatically Localized Weather Service

In the example composite service, shown in Figure 2.18, the user calls a service number in order to retrieve a weather forecast for the current location. We assume, that a weather forecast constituent service is available. It requires the location, for which the weather forecast shall be provided, as one of its invocation parameters. There is also already a service available that can provide the location of a user. Another component used in this composition is a user profile containing communication preferences of the user. It can be set, for example, through a self-service portal. The options are by SMS or by email. This preference can be retrieved using yet another service.

Combining all these constituent services allows the creation of a composite application that automatically requests the calling user's location, retrieves the weather for this location and ultimately communicates the weather forecast using the user's preferred channel. In case of email, a map with weather information is generated and added.

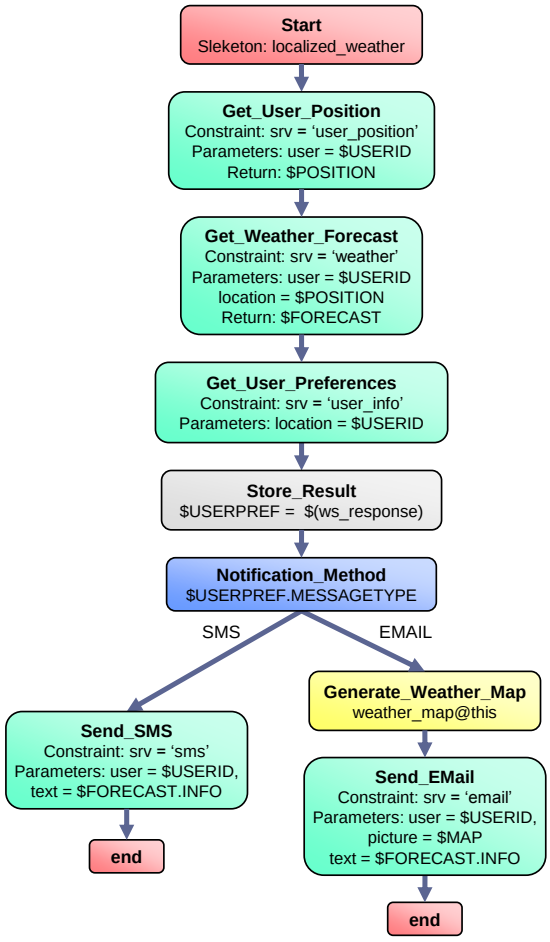

Figure 2.18: Implementation of a weather service application that automatically considers the user location and the preferred way of communicating the result

This service is only initiated through SIP. It does not compose any SIP constituent services. The used constituent services follow the stateless request-response scheme of, for example, Web Services.

Example: Blacklist and Whitelist for IMS Controlled Calls

Figure 2.19 shows a service composition that determines if a whitelist or a blacklist SIP service shall be deployed on the SIP session path for a user. A whitelist service would only allow calls to predefined addresses and a blacklist would block all destination addresses in the list. The user can use either of these services in his SIP sessions and the example composite application will add the respective service to the SIP service chain.

The criterion for using a whitelist service is that the user has configured this service. This is checked and if found true, the whitelist service is invoked. Invocation means that the SIP servlet container is instructed by the SIP CEA to include the service. If no whitelist

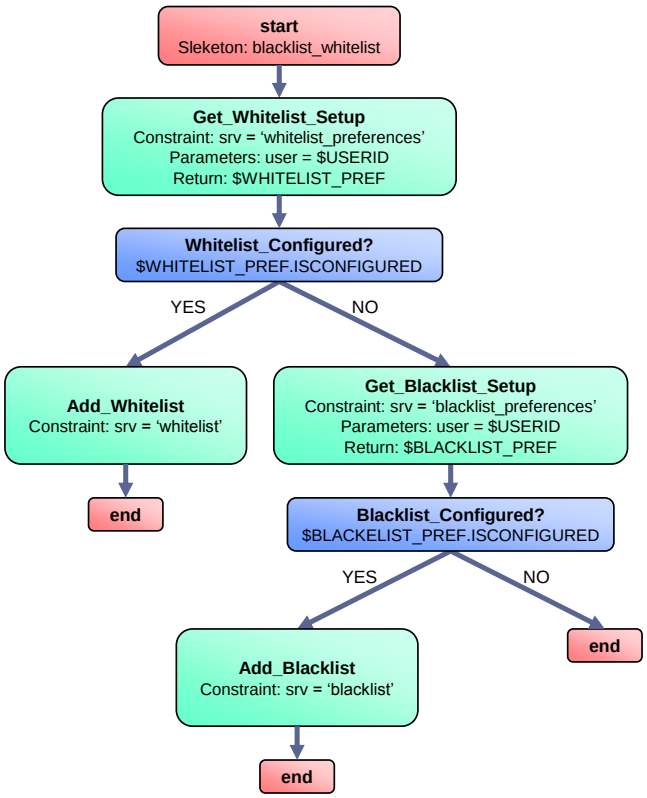

Figure 2.19: Implementation of a composite application that decides if a whitelist or blacklist service shall be on the SIP session path

service was configured, the configuration of a Blacklist service is checked and the service is added if it is configured.

This composition only instantiates either whitelist or Blacklist and never both. It also prefers whitelist if possible. In order to find out the configuration status of the whitelist and Blacklist related user data, non-SIP services are used. Whitelist_preference and Blacklist_Preference can, for example, be Web Services that query the user profile database looking for respective configuration data.

The service selection is for all service templates only based on constraints, thus, the service selection is agnostic of the technology the service is implemented in. The selection constraint is translated into an LDAP request towards the service database. In this example the parameter 'srv' from the service description is used in the constraint definition and request. For all service templates the service database has found a match. For 'Get_Whitelist_Setup' and 'Get_Blacklist_Setup' the selected service is a web service, and thus, it is executed using a SOAP request. For 'Add_Witelist' and 'Add_Blacklist' the selected service would be a SIP service, and thus, for the execution the SIP CEA and the SIP servlet container are used for their invocation.

In this example only one SIP service is selected and afterwards the skeleton ends and the

composition session is closed. The SIP session however still continues and is fully controlled by IMS. Thus, the selected SIP service continues to be active beyond the execution of the composition session. This composite application only deals with the concern of adding a blacklist or whitelist service. A Web Service was used in order to collect the data needed for taking this decision. For the same composition session there might however be further composite applications that add another set of SIP services.

Example: Family Call

The composite service shown in Figure 2.20 provides redirection of calls among family members. If the calling subscriber calls somebody from his or her family and that particular family member is busy, the call is automatically redirected to other family members until anybody from the family is reached. This type od service can be a useful offer for parents. When their child tries to call one parent, who is not available, at least somebody else in the family could be reached. If all tries fail an automatic SMS is sent to the parents.

The implementation of this example needs to react on subsequent SIP messages indicating that the called user is busy. This is done by using the constituent service 'Release Control'. It is a SIP service and implemented by means of the SIP execution agent. It indicates towards the SIP container that the skeleton has finished its composition task and the forward request can be sent on. Thus, the forward session setup by means of sending an INVITE proceeds towards the called family member. The composition session is on hold, waiting to be invoked again by a subsequent message.

If the called party is busy, a backwards SIP reply is sent through the session path indicating the cause 'busy subscriber' by code 486 'BUSY'. If the call is successful a 200 'OK' would have been sent as reply. When a reply is received, the composition execution resumes. In case of busy called party, the composition checks if there are further family members defined. If yes the SIP response is terminated and a new forward request is sent. This is done by the service 'Redirect Call'. It creates another INVITE forward request with address information of the identified next family member. The loop index is increased looping back to the release control service template. This means that the control is again given back to the SIP container after sending out the new INVITE, while the composition session goes on hold waiting for the next reply.

The composition finishes either when the call has reached one of the family members or when all family members were unsuccessfully tried. In the latter case, an SMS is sent to all family members that the called user has unsuccessfully tried to reach them before the composite application ends.

This example shows how the composition can react on subsequent messaging. It is doing this by going on hold while being subscribed to specific subsequent messages. Thus, the composition engine itself can listen on the SIP session path for messages that are of interest. In this respect it acts similar to a SIP application.

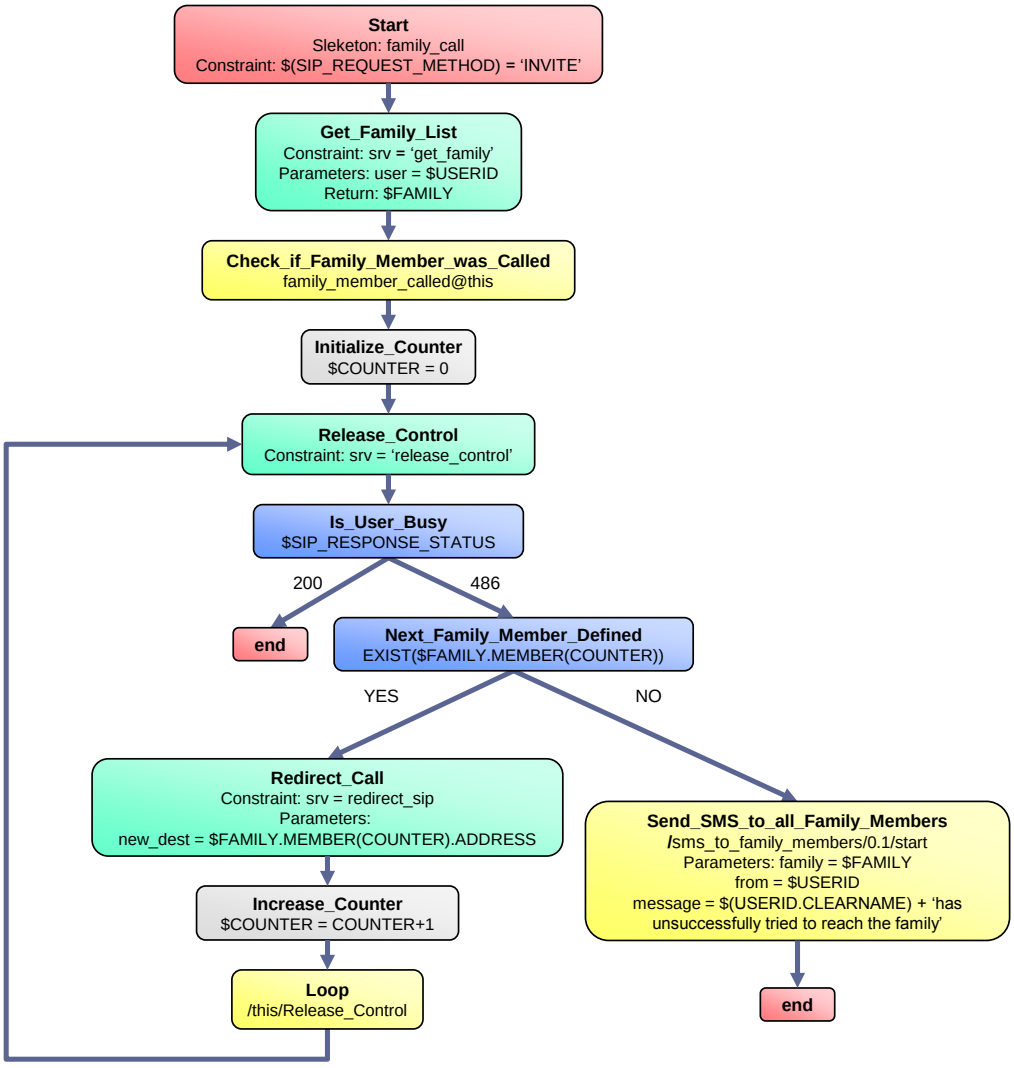

Figure 2.20: Implementation of a composition that automatically redirects call to other family members if the called family member is busy

2.3 Aspect Oriented Programming

This chapter provides a brief introduction and overview of Aspect Oriented Programming. This includes the basic ideas and objectives of aspect oriented methodology and an introduction to AOP specific terminology.

2.3.1 Cross-Cutting Concerns and Their Consequences

Software applications always implement a number of concerns. A concern is any kind of requirement or consideration that need to be addressed in the implementation in order to reach overall goals. This includes functional and non-functional requirements. The function an application needs to provide is as much a concern as, for example, execution characteristics, such as memory needs or execution time. In this sense any software system is first of all the embodiment of a set of concerns.

Concerns may be addressed outside the source code of the application. For example, execution latency requirements might be satisfied with the choice of a suitable execution platform. However, the concerns addressed by AOP and by this thesis are specifically those that have or even demand an implementation in software.

Concerns can roughly be distinguished in two main categories:

1. Core concerns capture the central functionality of a software application. They originate mainly from functional requirements.

2. Supplementary or secondary concerns capture additional supportive functionality or they originate in non functional considerations.

Core concerns are usually related to the function that is the main interest of the user and the reason why the user buys the application. In a banking application, the core concerns are, for example, customer and account management, interest computation, inter-bank transactions, ATM transactions. It is the central business logic of the application.

Supplementary concerns are often additional management functions required by the application provider. Authentication, charging and billing or storage management are frequent examples. Supplementary concerns are also the non-functional and system level considerations related to performance and resource usage of the application. Secondary concerns of, for example a banking application, are secondary concerns, such as logging, authorization, persistence, encryption and data synchronization.

An application is only suitable for productive use if all required concerns are addressed. Practically this means that a section of the source code is directly related to a particular concern. Often a comprehensive implementation of multiple concerns shows two problematic characteristics: Tangled code and scattered code, and frequently both at the same time.

Tangled code occurs if a single software module needs to manage several concerns at the same time. It is the result of putting together pieces of code addressing different concerns into one software module. Tangled code has some problematic characteristics. First of all

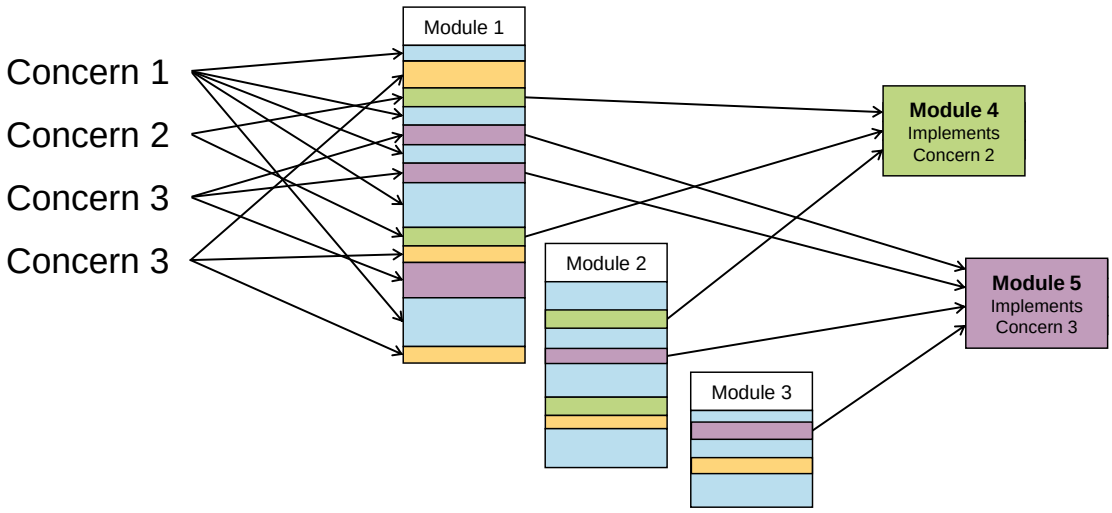

Figure 2.21: Code tangling and scattering

there can be a many identical code fragments that are repeated many times across the code base of the application. Thus, the volume of code is significantly increased.

A tangled implementation is also hard to reason about. The code structure is not explicit, and thus, the bigger picture of how a concern is implemented is not clearly visible because the code addressing one concern is mingled with code pieces addressing other concerns. Understanding and managing this kind of tangled source code is not straightforward. Changes needs to be comprehensively identified and consistently changed throughout the entire code base. Furthermore, changing the code for one concern must not break the implementation of others. Many distinct code location must be considered for a change and there is a strong inter-relation between the code of different concerns. This means, it is easy to overlook code fragments that also need modifications and it is absolutely necessary to understand all implemented concerns or a change that improves one concerns easily breaks another.

Scattered code appears if the code that implements a single concern is spread over several modules. This can either be the same code duplicated within multiple modules, or it can be different but complementary code spread over several modules. In both cases all code fragments together implement a function in order to address a concern. In a system using a database, performance concerns may, for example, affect all the modules accessing the database. This creates a strong dependency between software modules. Tangling and scattering are shown in Figure 2.21.

Separation of concerns by means of modularization of software is one of the most important tools in software development. It allows to split a complex development task into smaller and easier to handle pieces. As a result, work can be distributes across a development team and test team and also error localization and correction becomes easier. Tangling and scattering counteract a clean separation of concerns, and thus, directly effects efficiency in development, test and maintenance.

It is important to understand that scattered and tangled implementation is not caused by weak software design, and thus, it cannot be removed by optimizing the architecture and implementation. Fundamental functional interdependencies demand this kind of implementation, and thus, inhibit a clean separation of concerns into separate functional modules. This property of a concern is called cross-cutting, because the related source code might be distributed across the entire application. Cross-cutting impacts software design and development in many ways: poor traceability, low productivity, low code reuse, poor quality, and difficult evolution. All of these problems can directly effect the service provider's business result. Therefore, finding better approaches to software architecture, design and implementation can have a substantial business value. Aspect Oriented Programming is one viable solution that directly targets cross-cutting concerns by separating them into a modular implementation where conventional programming did fail.

2.3.2 The Basic Concepts of Aspect Oriented Programming

Aspect Oriented Programming (AOP) is a meta-programming technique. This means that it is a technique for writing programs that write or manipulate other programs or themselves. Programs are treated as data that allows to reason about the program logic and to apply reflective modifications.

The meta-programming technique that is understood today by the term Aspect Oriented Programming originated in the publication [12]. The author establishes an understanding of cross-cutting concerns and points out their relevance and effects on application development. Cross-cutting is an inherent property of complex systems. Nevertheless, cross-cutting concerns when considered in isolation have a clear purpose and a natural structure. Consequently, it is relatively clear in which modules and methods an individual concern needs to be implemented, when it crosses module boundaries, which resources it needs to utilize and what related flow of data is needed. AOP provides techniques to capture and express the structure of concerns in a modular way and it introduces linguistic and tool support.

The proposal in [12] is to keep code fragments that belong to a cross-cutting concern separate and implemented in dedicated modules. Thus, the core application only implements the core concerns resulting in lean and easy to understand code. A specialized engine is then inspecting the core application and the additional modules and creates automatically the tangled and scattered code that can be executed as usual. This process of assembling a complete application from modular concern fragments is called weaving. Consequently, the wanted application that implements all required concerns is not much different than without AOP. Its code is still tangled and scattered, but it is not the developer who has written this code directly. The idea is that the developer works with well separated concerns before the weaving automatically assembles the final application.

In [12] the function that performs the weaving is specifically written for a concern. Today, software development environments that offer an AOP methodology provide a generic and multi purpose configurable weaving mechanism. This means the weaving engine can be instructed to do weaving for handling many different kinds of concerns. This weaving en-

gine works with formal instructions that specify the logic of how the concern implementing modules need to be combined with and added into the core application.

The module that implements a supplementary concern is called 'advice'. The combination of an advice and the respective weaving instructions is called 'aspect'. The aspect therefore contains all ingredients that implement a cross-cutting concern. Thus, with the core application and the aspect available the overall application can be assembled.

Aspects are abstractions that can be used alongside other abstractions such as objects and classes. An aspect encapsulates functionality that implements a concern together with the detailed information where in the base application this function shall be applied. As a result, the base code stays clean in the sense that nothing of the implementation of the additional concern is visible in it.

The aim of Aspect Oriented Programming is not to replace procedural, functional, object-oriented or other programming models, but to complement them. AOP is also never used alone and an application cannot only consist of aspects. A core application that implements the core concerns is always needed and it is usually based on conventional programming methodology and languages. In the context of meta-programming, this core application is what AOP is reasoning about and where changes are applied.

Weaving instructions are rules for selecting those programming language constructs in the core application where advice needs to be applied. They are able to describe all relevant points in the source code, where advice can be inserted. These points are called 'join-points'. Weaving instructions are expressed using a weaving language that allows to specify the logic to identify relevant join-points in the core application. A weaving engine interprets these rules and applies advice accordingly.

The set of weaving instructions that define all join-points of an aspect are called 'point-cut'. Therefore, point-cuts are predicates that identify sets points in the execution of a program, where code of a cross-cutting concern needs to be added. Point-cuts define an observer pattern that determines what join-points to look out for. Given an aspect that modularizes a cross-cutting concern, its point-cuts serve as the interface between the cross-cutting concern and the rest of the system.

Join-points are not necessarily based on source code but they can in principle be formed from any information about the core application's operation. For example, events generated at run time by the core application can be used as conditions for executing advice code.

Aspects include a point-cut and advice, and therefore, a definition of where they need to be included into a program, as well as the code implementing a concern. Usually it not only necessary to identify where the advice needs to be included, but also how. This refers to the possibility to execute the advice before, after or instead of the join-point considering that the join-point itself represents a program statement or action.

There are several possibilities when weaving is applied. It might be possible already in the source code. This creates new source code with all contributions from core application and aspects. A weaving engine is doing this prior to compilation. This weaving engine might be implemented as part of the compiler, but in general, it is a separate process. The resulting intermediate source or byte code (in case of e.g. Java) can then be compiled and executed as one application.

Another possibility is that the weaving process uses byte-code as input. This means that aspects and advice are compiled separately into binary byte-code. The weaving engine then links the compiled units together forming a complete byte-code application.

Weaving at load-time means that aspects and base application are kept separate until the application is actually started. This means that any instance of an application can be different based on the weaving and aspects. This method allows for dynamic customizations based on the context of the application invocation. For different users different aspects might be applied.

All these methods offline, meaning prior to execution. Additionally there is the possibility to do online weaving. It weaves aspects on-demand at run time. This means, whenever a join-point is reached in the execution, the weaver triggers the additional execution of an advice. This means that the online weaver monitors and reasons about the execution of an application. The big advantage of online weaving is that run time data and state of the application can be considered in the weaving rules.

2.3.3 The AOP Enhanced Development Process

The key property of AOP in the development process is that it increases the possibility to separate concerns allowing for a higher degree of modularization. This potentially leads to improvements in all the usual benefits of modular design: Clarity, re-usability, increased code quality, easier to develop, configuration management, product line management, IP management and testing (modules separately). On business level the resulting impact is higher quality with less errors and primarily a shorter time-to-market and potential cost advantages.

Separation of concerns is a key principle of software design and implementation. It means that software is organized in such a way that each element in the program, for example a class, method or procedure, does one action and one action only. This allows to focus on implementing one element at a time. It also allows to understand each part of the program by knowing its concern, without the need to understand other elements. When changes are required, they are localized within a small number of elements.

The importance of separating concerns was recognized at an early stage in the history of computer science. Subroutines, which encapsulate a unit of functionality, were invented already in the early 1950s and subsequent program structuring mechanisms such as procedures and object classes have been designed to provide better tools for optimizing and handling the separation of concerns. Modularization of applications is therefore a fundamental tool in software development [51].

When designing a software based system from a well defined set of requirements the first step is to decompose the overall application into its components. An established strategy for this task would be by decomposing it by means of functions and sub-functions [51]. The resulting functional components are then implemented separately into distinct modules. This allows to separate at least those concerns that do not cross-cut. After all these individual parts have been implemented they are used to compose the overall application.

With AOP available, the decomposition - implementation - composition scheme works

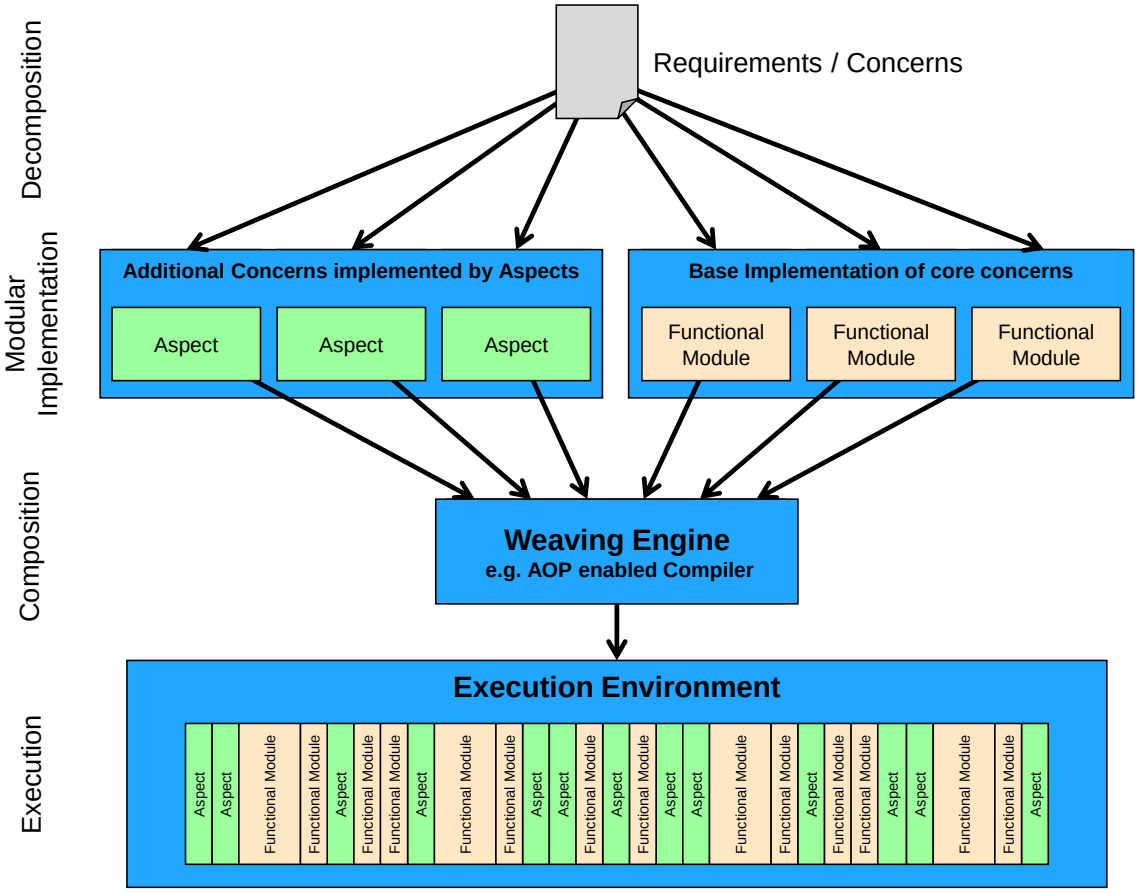

Figure 2.22: Decomposition and composition of an application using aspects

the same way. The important difference is however, that AOP allows to also decompose cross-cutting concerns into separate aspect components [52]. Doing so mans that the conventional modules become easier to implement because they have less concerns to consider. Instead aspects with their advice implementation and weaving instructions need to be implemented. The weaving engine contributes in the composition of all separately developed parts to form the application. This entire process is shown in Figure 2.22.

Decomposition of an application means finding a trade-off between simpler core modules and the additional need of designing point-cuts. The goal is to find the most efficient distribution of concerns between aspect and base modules. Strategies for this process are discussed, for example in [53] and [54]. There is a high amount of freedom in the decomposition process to decide which concerns are implemented as aspects, with effort spent on the weaving setup, and which are included within the base application, accepting some degree of tangling and scattering.

2.3.4 Introducing AOP for a Programming Language

Aspect Oriented Programming languages enable the isolation of cross-cutting concerns in aspects, with the advice in these aspects invoked at the appropriate points in the execution of the program. AOP methodology was made available as addition to many modern programming languages. The most prominent and most used example is AOP enabled Java called AspectJ. Also for other general-purpose procedural and object oriented programming language AOP is available. AspectC, AspectC++ and AspectPHP are, for example, AOP enabled variants of C, C++ and PHP.

When designing an AOP extension for an existing programming language the definition of a suitable join-point model essential. Each join-point model has to address the following questions [55]:

When can advice execute?
This question is related to the definition of what join-points are available and what elements of the underlying programming language can be used as join-point. To be useful a join point needs to be addressable and understandable by the programmer. It should also be stable across inconsequential program changes in order for an aspect to be stable across such changes. Lexical join-points are constructs in the text of the program code, for example keywords or class names. Dynamic join-points are run time actions, such as events or a specific invocation of a method that take place during execution of the program.

What are the means for identifying a join-point?
This refers to a language for defining point-cuts or weaving instructions. It is used to specify the conditions whether a given join-point matches, and thus, a related advice needs to be executed. Most useful weaving languages use a syntax that is similar to the base programming language syntax and it enables reuse through naming and combination. For example, the weaving language of AspectJ uses Java signatures.

How is code specified that is supposed to run at a join-point?
This refers to the question of how advice is implemented and how it interacts with the base application in the context of a join-point. It also refers to a method for identifying, selecting and addressing suitable advice. In many embodiments of AOP the advice is coded using the base programming language with unique identifiers for the advice module.

Join-point models are defined by specifying the set of join points that can be reached, how join-points are specified, the operations permitted at the join points, and the structural modifications that can be expressed. In this sense one of the main contribution of this thesis is the creation of a join-point model for a service composition language.

2.3.5 AspectJ and AspectWerkz

AspectJ [13, 14, 56] is the de-facto standard AOP language for Java. It is a general-purpose aspect oriented extension to Java and a typical example of an AOP framework,

where advice is weaved prior to execution [57, 58, 59]. Today AspectJ and related tooling is developed by an open source project within the Eclipse Foundation [60, 61].

AspectJ defines a few extensions to the Java language. It introduces, for example aspect, advice and point-cut as language constructs, thus, aspects and all of its components can directly be defined and used in the Java source code. Primitive point-cuts pick out sets of join-points and values based on a search pattern. User-defined point-cuts are named collections of join-points and values. AspectJ introduces two join-point models:

- A static join-point model where the join-points are member declarations and Inter-type declarations. The join-point is identified by type patterns and signatures. Those are declarations that cut across classes and their hierarchies. The execution semantics is affected by the definition of new members.
- A dynamic join-point model based on point-cuts and advice, where join-points are well defined points in the dynamic call graph of an application. For example, a method call is considered to be a join-point. Point-cuts are used for identifying the join-points and advice is used for modification of execution semantics.

Inter-type declarations allow to apply cross-cutting changes across the class hierarchy. They therefore extend the functionality of the construct that is already present in the base application. They are used to add a new capability to multiple classes. This allows to declare and implement the new capability, for example by adding new methods, fields and interfaces. Instead of defining this new method in all classes it is only defined once and added to all effected classes by means of weaving. This weaving process is done statically. This means it is entirely done prior to compilation.

In the following example an aspect is defined that adds the new method 'additionalMethod' to the class 'ExampleClass'

```
Aspect ExampleAspect {
    void ExampleClass.additionalMethod (Variabletype variable) {
        ...
    }
}
```

The point-cut and advice join-point model targets the Java program rather than its class hierarchy. Many points in the execution of the Java program can be used as join-points. Although any identifiable point in a program's execution is in theory a join-point, AspectJ limits the available join-points to those usable in a systematic manner. Typical examples are a method and constructor call and execution, read or write access to a field, object and class initialization or exception handler execution. The point-cut is defined by providing a search pattern that captures the declaration of, for example, all method calls that shall be effected. For example, the following point-cut definition

```
pointcut set() : execution(* set*(..) ) && this(Point);
```

matches the execution of any instance method within an object of type 'Point' whose name begins with 'set'. The search patterns for join-points can be quite powerful when using wild-cards.

Advice definitions complement the point-cuts. They specify the code that is supposed to be executed at the join-point. It is possible to define before, after and around advice. These advice types differ in the order of execution. Advice with a 'before' declaration is executed before the join-point. Respectively, the 'after' advice leads to execution of the base actions associated with the join-point prior to the advice code. For example, a method call that constitutes the join-point is executed and needs to return, before an associated advice is invoked. The 'around' advice type execute only the advice code. This can be used to inhibit the join-point execution. There is however the possibility to instruct explicitly from within the advice to proceed with the join-point execution. Point-cuts also allow the exposure of context at the join-point to an advice.

The following example advice can be used in combination with the point-cut defined above. It initiates, that the data object is updated every time something on 'Point' is set.

```
after () : set() {
  Data.update();
}
```

The AspectJ weaver is an aspect compiler allowing the weaving to be applied in many different ways either on source code level or on Java bytecode level. In all cases the final application created by the AspectJ compiler is pure Java byte code. It can run on any Java virtual machine. This weaver is accompanied by supplementary tools such as an aspect aware debugger and IDE integration.

AspectWerkz is a dynamic, lightweight and high-performance AOP and Aspect Oriented Software Development (AOSD) framework for Java. It has been merged with the AspectJ project, which supports AspectWerkz concepts since AspectJ version 5. Unlike AspectJ, prior to version 5, AspectWerkz did not add any new language constructs to Java, but instead it supports declaration of aspects within Java annotations. In other words, there is a set of custom annotations that expresses an AOP language. Aspects can be written by using these custom annotations in the base code. Weaving is done by modifying bytecode on order to weave classes at project build-time, class load time, as well as run time. In order to do so AspectWerkz uses standardized Java virtual machine level APIs.

2.3.6 Dynamic AOP

Aspect Oriented Programming can be classified into two categories with respect to the time when weaving is applied: Static AOP systems weave the aspects at compile time or load time. Dynamic AOP systems weave the aspects at run time. AspectJ is a typical example for static weaving, which is often referred to as offline weaving. Statically woven aspects cannot be removed or modified any more at run time.

Using dynamic AOP based on online weaving can result in substantial benefits, if a concern crosscuts over many modules and potentially changes dynamically at run time [62].

A good example for this kind of concern is the correction of system errors that constitute a security breach. Critical corrections like this need to be rolled out as fast as possible and broadly across all effected applications. Another example might be policy changes. Also analytics and application monitoring concerns that are only temporarily needed would a good example where dynamic AOP is beneficial. The example scenario is an application that shows deviations from normal behavior. Support personal can then activate analytics and debugging functions by applying them as an aspect in order to view and reason about run time data of the application. This aspect can be removed when not needed any more in order to save the extra execution capacity. It is possible without stopping and re-starting the application. In the same example scenario the monitoring might have detected an error. This can be fixed immediately also without an application re-start using another aspect for emergency error correction.

Another example where dynamic AOP might be useful is a system where it is beneficial to switch between decision strategies dynamically. An example are response caches of method calls [62]. First of all this kind of caching function cross-cuts the entire application because it needs to be applied at multiple method calls. However, different caching strategies might be implemented using specific sets of aspects. There is no one single caching strategy that would be optimal for all kinds of web applications. A system based on dynamic aspect weaving could dynamically switch between strategies by replacing the used set of aspects. It can therefore react dynamically on low cache hit ratios with a different caching strategy. Traditional design patterns of object-oriented techniques never modularize such cross-cutting concern, and even less switch it at run time. Static AOP would not work well for this example. It would need to weave in all caching strategy aspects together with the strategy selection code at each join point. This implies many extra checks to be executed, and thus, a potentially substantial performance burden. With dynamic AOP only one caching strategy aspect is applied at a time and the reasoning about cache performance and the replacement logic is outside the application.

Dynamic AOP can be reached by inserting aspects into the application by dynamic code translation or it can use the Java debugger to execute advice at break-points or method calls [62]. This means there are two weaving strategies available and they can be chosen depending on which will provide better performance.

JBoss AOP is an AOP implementation that enables the use of AOP in the JBoss application server [63]. JBoss is an open source JEE application server. It is an independent framework, in this respect similar to AspectJ, that can be used with any Java program.

The most interesting feature of the AOP implementation of JBoss is the concept of interceptors [64, 65]. Interceptors enable the system to transparently add behavior provided by advice services into any object. This means aspect weaving is applied dynamically at run time rather than at design time or at byte-code level. Being performed at run time, the weaving process of JBoss AOP can consider run time data. The result is a highly dynamic weaving environment. JBoss AOP does not extend Java. Advice is realized by means of new Java classes.

Another approach for dynamic weaving is based on dynamic method wrappers that allow advice code being inserted around a method body [66]. It uses Java mechanism to

dynamically add wrappers into binary libraries.

[67] investigates static and dynamic weaving based in AOP extensions to Smalltalk. This study concludes that it is beneficial to have a weaver that allows static and dynamic weaving combined. It concludes that all aspects that do not need dynamic adaptations at run time should be statically weaved for performance reasons. [68] introduces an AOP mechanism in Java that introduces aspects as extensions of a class called 'Aspect'. For dynamic weaving at run time a modified Java Virtual Machine is proposed. If a join-point is reached while interpreting the application, additional code is executed by the virtual machine that manages the weaving of aspects. Furthermore the performance of dynamic weaving is discussed. The authors find that overhead at each join point leads to considerable decrease in execution performance. Both results are highly relevant for this thesis as service performance is one of the key concerns for telecommunication applications while online weaving would enable features in aspect handling that re highly interesting for telecommunication service environments.

Dynamic AOP or dynamic weaving is often referred to as online weaving while static weaving is often also called offline weaving. Please note the difference in terminology of dynamic and static weaving as described here and dynamic and static join-points as described in Chapter 2.3.5.

2.3.7 AOP for Business Processes

Middleware constitutes feature-rich software. This implies that it implements many concerns [69]. Consequently, this causes a high degree of complexity, because many of these concerns are cross-cutting. In this context, middleware is defined as a software that mediates between two separate and often already existing programs. Application servers are a typical example of middleware. It has the following characteristics:

- Many applications from different vendors can utilize a middleware component.
- Middleware potentially incorporates many policies.
- Middleware is typically customizable in order to accommodate a great variety of applications.

A composition engine shows these characteristics and is therefore yet another example of middleware. What follows is that composition engines are a good candidate for using AOP. Consequently, next to traditional programming languages, AOP concepts were developed for the languages used in business process definition and execution. The main target of these efforts was BPEL.

[70] demonstrates how business processes can be extended by means of AOP. The idea is to add additional process steps into the work flow or to redefine existing process steps. The base of this work is AspectJ. This means the process is implemented in Java rather than a specialized business process language, such as BPEL and BPMN. Nevertheless,

it demonstrates the idea using aspects to target the main building blocks of a business processes in order to redefine the process.

[71] analyses generic needs of an AOP system for work flows. In particular it proposes modification actions to the work flows that could be implemented by means of aspects. It discusses fundamental operations to modify a work flow by means of, for example, replacement of an activity, addition of an activity, adding an additional thread, converging multiple threads into one and addition of loops for repetition of an activity. A framework for realizing these types of dynamic changes needs to have reflective capabilities. Furthermore it argues that dynamic weaving at run time would allow to apply improvements to the process with the need to restart it.

JasCo

JasCo [72] is a language for Aspect Oriented Programming in the context of component based software development (CBSD). CBSD refers to developing software applications by assembling pre-produced and independently deployed components. Each of them delivers a specific service as contribution to the overall application [3]. Also this development paradigm suffers from cross-cutting concerns across components. JasCo introduces Aspect Oriented Programming specifically for component based software development.

JasCo is an aspect oriented extension to Java. It introduces two new concepts: JasCo beans and connectors. JasCo beans is an extension to the Java Beans [73] component model introducing an aspect enabled component model. An aspect is in this respect a Java Bean that is able to declare a number of hooks. These hooks are special classes and they specify when advice shall be executed together with the base code. Thus, a Java bean with hooks is the equivalent of advice and point-cut when comparing it to AspectJ. The main difference is that hooks are generic in the sense that their definition is not tied to a specific context. This keeps the JasCo bean re-usable in many different application contexts. Consequently a reusable aspect component can be designed.

With hooks in JasCo beans it is already possible to define when advice shall be executed. JasCo connectors allow to add the information where this aspect shall be applied. The connector deploys the context-less JasCo aspect within a context where it shall be used. The connector therefore completes the point-cut. This separation of point-cut definition between a generic part in the hook definition and the context aware part in the connector allows flexible and dynamic deployment of JasCo beans as component in various applications.

Using the JasCo infrastructure the decoupling of web services from the client application was demonstrated [74]. This is done in order to reach a dynamic selection of the web-service that instantiates the service needed by the client application. For this purpose a service management layer is introduced between the client application and the web services. This additional layer captures a generic request for a service, selects a suitable web-service and translates the generic original request to a concrete request to the selected web service. This behavior is reached by using redirection aspects that define the logic for intercepting certain client application requests and for replacing them with web-service invocations.

These aspects therefore encapsulate all details of this selection and mediation process. This web service selection process is done transparently for the client application.

This use of aspects for added flexibility in web-service selection is similar to the constraint based service selection of the Ericsson Composition Engine. The main difference is that selection constraints and the related dynamic service invocation process are native part of the composition and execution model of the Ericsson Composition Engine, and thus, also an integral part of the composition language. The method presented in [74] reaches this transparently, and thus, no additions to the composition model of the client application are needed.

AO4BPEL

AO4BPEL [75, 76, 77] is an AOP method and framework for business processes defined in BPEL. In AO4BPEL each BPEL activity is a possible join-point. Because BPEL processes are expressed as XML documents, the point-cut is based on XPath expressions. This way all activities are selected where additional cross-cutting functionality shall be executed. This means aspects are applied to the XML representation of BPEL. AO4BPEL supports weaving before, after and around a BPEL activity.

The advice itself is also an activity defined in BPEL. Both, offline weaving and online weaving have been implemented based on BPEL this concept. Furthermore, the implementation of transaction compensations were demonstrated using aspects [78].

BPEL'n'Aspects

BPEL'n'Aspects [79] is an approach to use aspect oriented design together with BPEL [6] processes. The central idea is that aspects can be weaved into the process execution at each step in which the process execution engine interprets the process model. More specifically before and after each step in the process execution additional advice can be executed. The triggering of advice execution, and therefore, the base of the join-point model are events in the execution engine.

BPEL language elements serve as join-points. Advices are web-service operations, thus, any web-service can be an advice. Point-cuts are done by means of subscribing to events in the process execution. These events indicate the execution of a certain BPEL language element in combination with run time context. Consequently the Aspect is a combination of event subscriptions and web services. Weaving is done by dynamic online observation of event subscription and respective invocation of web services.

Based on BPEL'n'Aspects a solution for transactional compensations that includes the executed aspects is proposed [80, 81]. This is an important feature for long-running business processes.

The approach of BPEL'n'Aspects to Aspect Oriented Programming introduced for a business process execution language bears some similarities to the Aspect Oriented Programming approach that is introduced in this thesis. For example, the idea to use events of the execution engine as weaving triggers is similar. The underlying process model and

the language used for describing the process is different. In contrast to typical BPEL engines, the Ericsson Composition Engine supports not only the execution model of web services, but a broad range of service technologies including SIP services, where the roles of services and composer are significantly different compared to BPEL environments. Furthermore, the Ericsson Composition Engine uses a constraint based dynamic service selection mechanism. The resulting AOP solution with the Ericsson Composition Engine considers these important features. However, transactional compensation is not considered within the scope of this thesis. The reason is that the composite applications that are typically done in telecommunication service scenarios are not long-running business processes, but services provided within communication sessions, where compensation activities are not equally essential.

Chapter 3

Aspect Life-Cycle and Management

Since Aspect Oriented Programming was proposed in 1997 [12], it has become a tool available to developers when facing cross-cutting concerns. Starting with an Aspect Oriented Programming framework added to Java [13, 14] respective language extensions and tooling became available for most popular and most widely used general purpose programming languages.

Applications and service implementations used within a service provider's business context usually undergo a life-cycle with well defined phases and related development, maintenance and decision processes. The respective underlying life-cycle model anchors the service application and its environment within the business processes of the enterprise that owns and offers the service. These life-cycle models help to understand responsibilities, plan service related actions and, in general, facilitate communication between all parties involved in, for example, development, marketing and operation. In this respect it is highly important to understand the role and handling of aspects within an application life-cycle and development process.

3.1 Service and Aspect Live-Cycle

This thesis puts a special focus on telecommunication services and a service composition technology that is particularly developed for implementing service applications for that domain. The used composition technology uses constituent services that abstract telecommunication services by applying SOA principles as described in Chapter 2.2. Due to the SOA heritage the life-cycle of the resulting services is similar to the well known and well understood life-cycle of SOA web services. Consequently, their life-cycle model is a sensible choice within the scope of this thesis. A generic life-cycle model for service-oriented design and development as defined in [4] is used here as base for discussing telecommunication services, composite applications and also the potential roles of aspects. The model defines six main phases shown in Figure 3.1.

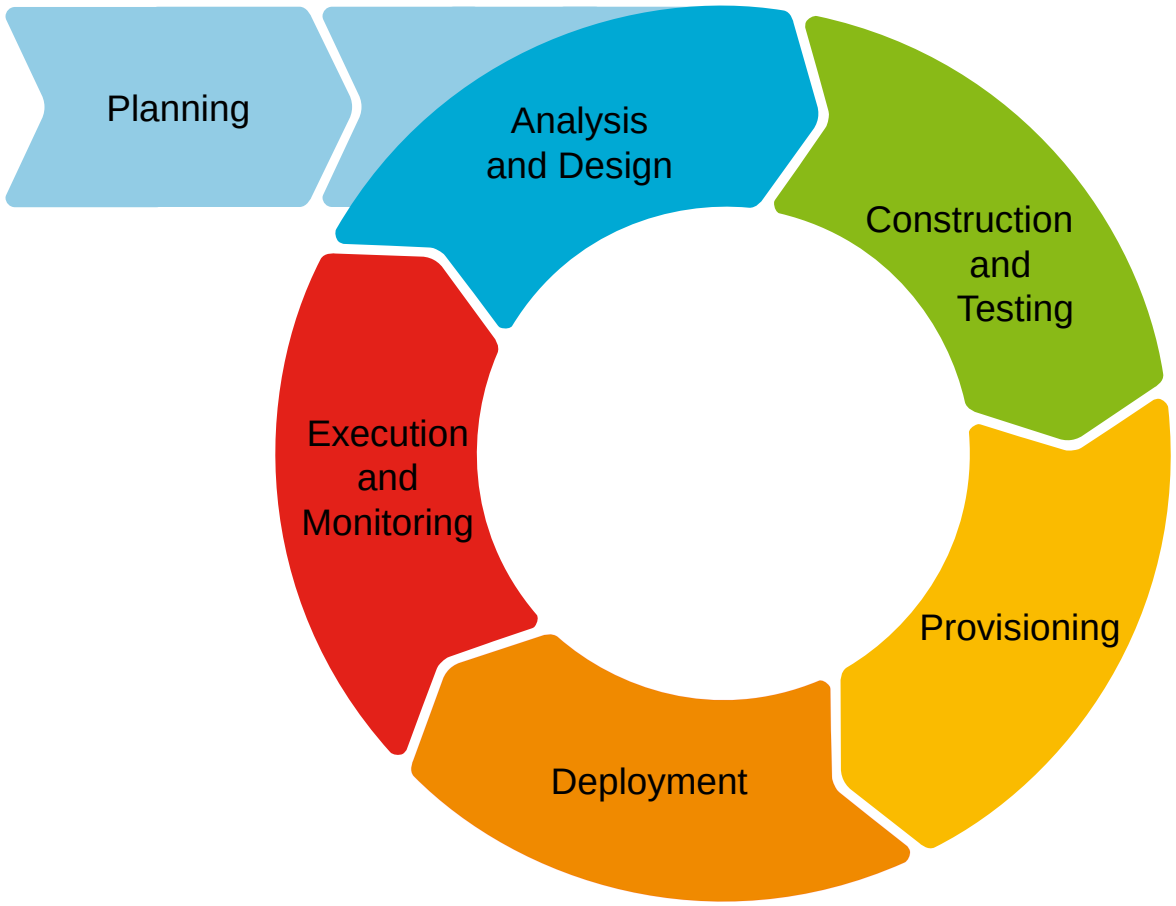

Figure 3.1: SOA service life-cycle

Planning

The planning phase is a preparatory activity. In this phase the scope of the service application is determined in the context of, for example, the general goals of the enterprise and the demands of the targeted customers and market. Here, it is primarily essential to understand the business needs. Furthermore, financial and organizational constraints need to be analyzed in order to define the right parameters for the following phases of the service life-cycle. This effort ideally results in well defined goals and concerns expressed in high-level requirements. Tools, such as the business model canvas [82], can be used in this phase as a methodology for gaining necessary insights into all influencing factors of the business case related to the wanted service offer.

In a telecommunication environment the driver behind change in the service landscape is often not individual innovation in specific end-user services, but rather large scale technological evolution of the entire industry. A good example for this is the transition from 2G/GSM cellular networks towards 3G/UMTS and further on towards 4G/LTE. These

changes are often driven by standardization and regulation leading to a large number of individual services to be implemented or upgraded at a certain point in time. A historical example, which demonstrates this process, is the evolution of CAMEL services. Interoperability across systems from different vendors demanded detailed standardization of the CAMEL infrastructure and related protocols. A major standard release called "CAMEL Phase" establishes a set of common functions and protocol extensions within nodes and protocols. This introduces new capabilities that directly facilitates a new set of service offers for the end user. Without the interoperability from new standards being widely adopted, the new services would not be possible.

New services being tied to widely available standards means that many of the them become a commodity soon after the standards are released. The business strategy with respect to this type of services is to a great extent dominated by speed of market introduction once standards are finalized and respectively upgraded service networks are available. For this reason the telecommunication industry is often characterized by races to introduce new technologies. System vendors race to be the first offering the new standardized capabilities in their equipment offers and operators race to be the first on the market utilizing the new capabilities in order to win a slight edge in attracting customers.

Next to the standardization driven innovation cycles, added value services are used to create an additional competitive edge. These services distinguish an operator from its competitors through exclusiveness and unique capabilities. Both types of innovation cycles lead to different types of criteria used in the planning phase, but the same life-cycle model can be applied. It is generic enough to describe a great variety of cycle times and it scales from describing small applications up to complex service and process landscapes.

Analysis and Design

The planning phase is succeeded by the analysis phase. In this phase all information that was previously prepared and collected is used in order to identify the detailed requirements of a SOA-based service implementation. The high level outline of the service offering is defined. This includes the definition and layout of the needed business processes and the business services used within. Guiding criteria can, for example, be the potential to reuse existing processes and services, their expected business impact, organizational capabilities and technical viability and feasibility. High efficiency can be reached when assembling the new service application offer mostly from already existing business services and processes. In this respect gaps in the existing service landscape need to be filled with newly designed services in order to get all components in place that are needed for the desired overall service application.

The processes and services described in the analysis phase are abstract entities. In the following design phase they are transformed into a set of concrete services represented by their service interface description. This concept of SOA style service descriptions was introduced to telecommunication services as part of the work on the Ericsson Composition Engine. This was the essential step that makes the SOA life-cycle model and the related development methodologies applicable to typical telecommunication services although their

technological and implementation details can differ significantly from the SOA typical web services.

Through the interface definition the design phase establishes the foundations for many SOA characteristics. Here, the granularity of services is defined. Closely related are reusability and composability of the constituent services.

The design phase also establishes details about service structure and wanted behavior. This includes, for example, the protocols that need to be used and the underlying technology on which the service shall be implemented. For example, an IMS service needs to understand SIP and the related protocol state machine that enables it to function within an end-to-end session context.

Construction and Testing

In the construction phase the physical realization of the needed services is added. The services are implemented choosing a suitable programming language and methodology. This phase also includes the choice and setup of a respective hosting and execution infrastructure. Application servers, databases and composition engines are selected. These assets might already be available and in use for other service offers following a general technology policy. Nevertheless, gaps need to be identified and filled, for example a capacity shortage once the new service application is additionally hosted by existing infrastructure. Another example might be essential software tools needed for the new application that might still be missing in the available environment.

Construction goes hand-in-hand with testing. It verifies all functional and non-functional properties required from the service application.

Provisioning

At the end to the construction and testing phase the required service application is fully implemented from programming point of view. Service provisioning then creates the technical and business context in which the service will be provided to customers. An important concern of provisioning is governance. It establishes ties between the application and various operational units within the enterprise. Especially the integration of the new service application into the business functions of the organization is important in order to reach well controlled business assurance. A typical example is the setup of rating and billing models for the service usage. Especially for telecommunication services the question who is billed for a service usage can be complex and a central objective of business support systems.

Business-to-business service offerings are often governed by formal service level agreements with agreed quality of service levels. This is done by specifying detailed target metrics and well defined compensations in case the metrics are not met. Setting up an environment for controlling the related key performance indicators and for handling contractual consequences is also a task of provisioning. External certification is in this respect a helpful supportive activity for establishing trust in the capabilities of a service offer.

Management and operation functions need to be in place in order to control the service offer while being used. In telecommunication networks Service Operation Centers (SOC) are centralized units for consolidated and cost efficient operation of entire service networks with the responsibility to monitor the entire technical operation with fast reaction times in case of incidents.

Deployment

The deployment phase executes the roll-out of the new service. This includes that service descriptions are uploaded and published through a service registry enabling others to find the new service and bind it into applications. Furthermore the service application run time is installed on application servers within the hosting infrastructure. The respective invocation end-points are also published in order to allow clients to instantiate and use the service. While the service interface description is abstract, the invocation interface mainly depends on the chosen service technology. In SOA environments the client usually deals exclusively with web services, while many telecommunication services demand other specific invocation interfaces and procedures.

Deployment also means that all operational and business support systems are updated in order to fully integrate and enable the new service offer. For example, product catalogs are updated and order and fulfillment processes might be changed and staff is trained in order to become aware of the new service. Furthermore, the needed modifications are not necessarily only internal, but can include suppliers and other external partners. In order to preserve a consistent operation the timing of deployment phases and actions plays an important role and needs to be well planned.

Execution and Monitoring

With all deployment activities finished, the execution phase is entered. It is the phase in which the service becomes available to customers. This technically means that client processes can find and bind the service interface and then invoke a new instance of the service. If the service itself is implemented by a composite application, it is executed by selecting, binding and invoking further constituent services. Please note that the Ericsson Composition Engine performs a full data driven selection of constituent services at run time prior to binding and invocation. Furthermore, other than stateless and request-response based service invocation and usage procedures are possible with the Ericsson Composition Engine when using IMS/SIP services as part of the composite application.

The execution of processes and services needs to be monitored in order to detect if they do behave and perform as expected. This monitoring is a key feature for operation and business assurance, because it is the technical enabler for corrective actions or compensations in case service level agreements were broken. The collected monitoring data can also be used in a retrospection process of the service's value and performance at the beginning of the next iteration of it's life-cycle.

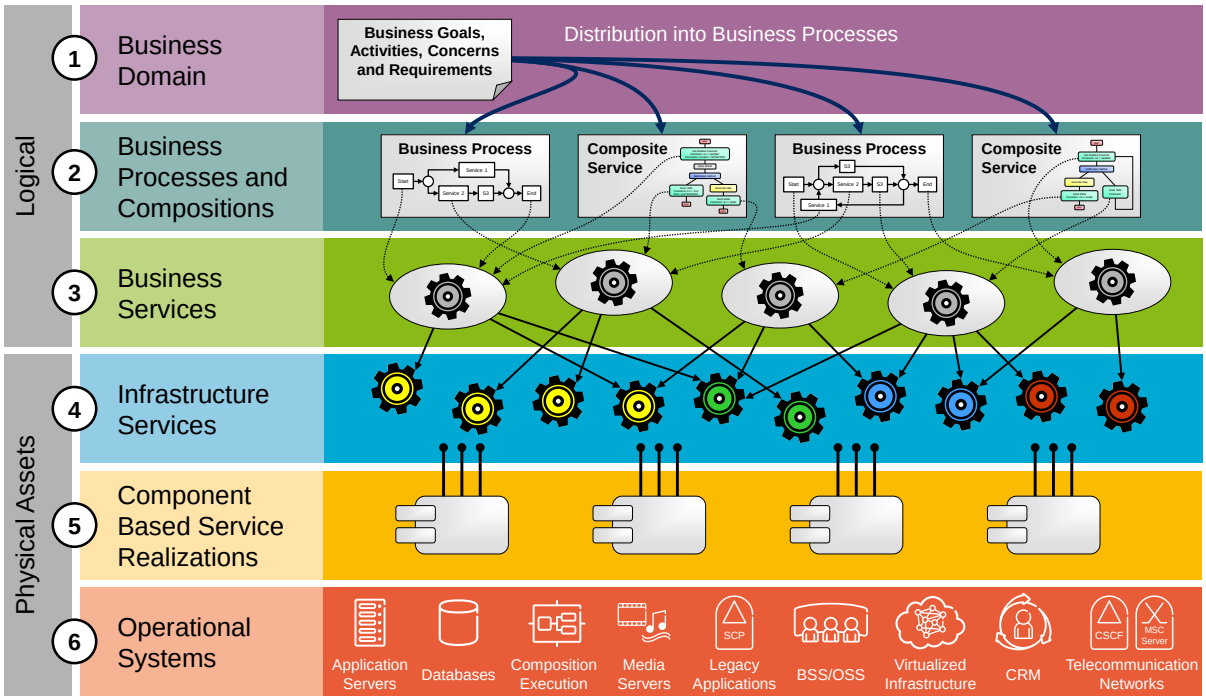

Figure 3.2: SOA Layers of functional abstraction

Further Life-Cycle Iterations

When the cycle is started over again the new analysis phase has now additional information from the previous cycle. For example, key performance indicators were collected from monitoring of service execution. At this point also financial data is available describing the related business performance. Furthermore new insights into market and competitor situation might be available and new business goals might be defined. All this information can first of all establish the need for starting a new iteration if the life-cycle. This leads to revised technical and business requirements and ultimately to changes in the service application design. The changes are implemented in a new round of construction, testing, provisioning and deployment until a new version of the service is ready to be used replacing or complementing the older version.

3.2 Layers of Abstraction in a SOA

The potential to reuse constituent services in many different application contexts is established mainly through abstraction. Many levels of abstraction are used between the concerns of the business domain and the detailed implementation of the programs behind service offers. These levels of abstraction are closely related to the service life-cycle. Proceeding through the life-cycle for creating and offering a service also means to go through the levels of abstraction from top to bottom.

The model of abstraction, used in this thesis, contains six layers shown in Figure 3.2. It is based on a general model for a SOA environment [4]. Only minor adaptations were necessary in order to allocate the Ericsson Composition Engine based orchestration and telecommunication networks based services and use cases within the abstraction model.

The first 3 levels in the abstraction model are concerned with breaking down the business domain into its business processes and atomic but still abstract services. For each of these atomic services and for service applications implemented by means of service composition, a life-cycle according to Chapter 3.1 is defined. The abstraction defined on levels 1-3 becomes subject to the analysis and design phases in the life-cycle model. The transition from atomic business services towards their implementation based on infrastructure services corresponds to entering the construction phase of the life-cycle model.

Level 1: Business Domain

The business domain refers first of all to functional domains within an enterprise. An enterprise can be partitioned into several disjoint domains. Typical examples within a mobile network operator and service provider are network operation, marketing, development, finance, customer care and human resources. Often the business domains are directly reflected in the internal organizational structure of an enterprise. The operation of a functional domain can be described by a set of business processes. They embody an abstract description of goals and activities leading concerns and requirements for subsequent layers in the abstraction model and for decisions taken in the related service life-cycles.

Level 2: Business Processes and Service Compositions

On this level the business processes are outlined in detail defining an abstract view on all activities within an entire enterprise or just within a business domain. This is done by specifying the detailed rules that determine when, how and which services shall be provided and consumed. Languages, such as BPEL, BPMN and the skeleton language of the Ericsson Composition Engine, were designed to provide the needed expressiveness and semantics.

Business processes rely on the availability of the right services to be orchestrated. However, a constituent service of a business process can itself be implemented as orchestration of yet another set of constituent services. Multiple levels of processes and sub-processes are created by dividing and decomposing the functional requirements of business processes into smaller functional units. This is done until finally a level of granularity is reached where further sub-devision is not feasible any more. The result is a set of singular business services.

Level 3: Business Services

Business services are the atomic constituent services of business processes. Ideally they are reusable in many different business process contexts. This is reached by finding a level

of granularity where each business service provides a generic function that represents a generic business task. Examples are creating an order, sending an invoice, notification of the user, setting up a call or deliver a video stream.

Business services are defined by their interface and service description. This makes them abstract assets that exist independently of a particular implementation.

Level 4: Infrastructure Services

Infrastructure services are the physical implementation to abstract business services. All assets on levels 1-3 are based on abstract descriptions of processes, services and service interfaces. Starting at level 4 the business services are implemented by using infrastructure services. While the business services can, to a great extent, be defined and used without a certain technical environment in place. Infrastructure services directly utilize and present the available functions and capabilities of the used technical infrastructure. For example, a business service defines access to user data. The infrastructure services used to implement this data access deals with the details of how and where the data is stored and what protocols and logic needs to be used for getting to it.

Level 5: Component Based Service Realizations

Components are often autonomous software units that exist within the implementation of infrastructure systems and middle-ware exposing lower level functions. The implementation of infrastructure services is based on a combination of these functions.

Level 6: Operational Systems

Operational systems are the enabling applications in the infrastructure. Examples are application servers and composition engines that execute service implementations and compositions, Databases and file-systems that handle application data and media servers that provide streaming content. Application hosting environments can be used that are, for example, based on a virtualized infrastructure. The telecommunication network with its diverse service and control nodes is also allocated on the operational systems layer. Also complete application suits can be used, for example a customer management system or a BSS/OSS solution. Legacy applications that were not implemented with a SOA approach would also be considered to be an operational system.

3.3 Aspects in the Life-Cycle of Service Applications

Aspect Oriented Programming was introduced as a technique to directly aid the implementation of software applications. It is therefore closely related to and used within the context of the chosen programming language. This results in a tight coupling between the base application and the additional aspects. The aspects are often not much more than

yet another internal unit within the code base of the application combined with a set of weaving instructions. From life-cycle point of view this means the following:

1. Aspects are defined and applied in the construction phase when creating the technical implementation of the services and applications.

2. Aspects are bound to services throughout the service life-cycle. They do not have an independent life-cycles.

This is in particular true if off-line weaving is used because the aspects and the base application become an indistinguishable unit after the built is performed. For the layered model of service abstraction this means that aspects are allocated where their base programming language is used for concrete implementations. This usually refers to the lower layers 4-6.

Introducing Aspect Oriented Programming techniques in service composition and orchestration means that it can already be used in the design phase of the life-cycle. In this phase the distribution of functional and non-functional concerns and requirements into business processes and abstract business services is performed. Thus, using AOP for service compositions and business processes allows to deal with some cross-cutting concerns before actually constructing the services. The result can be a more modular set of needed business services. The construction phase then needs to deal with less complexity due to some cross-cutting concerns that were already considered.

In the layered abstraction model, this lifts aspect up to the layers 1-3 where the languages for business process modeling and definition of service compositions are used. AOP capabilities can therefore already be used in the formal definition of business processes and the specification of business services. Additionally further aspect oriented techniques can be applied to the implementation on lower layers based on the programming methodologies and languages used there.

The requirement to actively assure that service level agreements are kept can be used to demonstrate the differences between using aspect oriented methodology in the design phase or not. The wanted functionality is that the overall execution time of a composite service shall be kept below a predefined limit. This shall be enforced by dynamic selection and configuration of constituent services. The idea is that a longer than expected execution time of one constituent service is compensated by a more efficient configuration of constituent services, which follow at a later stage in the composition execution. This new configuration might come at a higher price due to using a particular higher performance constituent service. The goal is to only use the faster and more expensive service configurations if needed to compensate slow execution and to ultimately avoid penalties for not fulfilling the SLA.

This optimization of service selection is a cross-cutting concern that is already apparent on composition level. In a non AOP based implementation this would mean that the requirement to implement this feature is handed down to the level of constituent services. A possible solution would be to implement a central control service that does the bookkeeping

of the critical overall execution time. This central control service would also decide on the need of compensatory actions. Each constituent service then needs to implement an interface to communicate with that central control service. On this interface timing information is reported by the services and faster execution might be requested by the central controller. This additional functionality would be needed across all implementations of suitable candidate constituent services. Consequently not only the implementation of new services becomes more complex due to the additional interface and function, but also the re-use potential of already existing services is drastically limited.

An aspect oriented implementation of the composition could avoid these problems. Still, a new central controlling service would be needed, but the constituent services would not be impacted directly. The re-configuration of them would be implemented by means of aspects. Advice code would communicate with the central controller. This advice would be weaved into the composition at every service invocation. It can then either change invocation parameters or replace a service binding by selecting another more efficient implementation or configuration of the service.

This example shows how aspect orientation is applied as early as possible in the service life-cycle and how that can help to reduce complexity in later phases. It ultimately leads to a cleaner SOA implementation with respect to reusability of services.

3.4 Dynamically Assigned Aspects with Individual Life-Cycles

The use of Aspect Oriented Programming and an aspect oriented methodology in software development as described in Chapter 3.3 usually ties aspects tightly to their base application. Only the design and construction phases are subject to implementing and managing aspects. In later phases after the aspects were created they become an inseparable unit with the base application throughout the application's life-cycle phases.

Aspect Oriented Programming can be used differently creating more flexibly and separating the life-cycle of aspects from the service applications. Aspects have the potential to modify a base application after it has left its construction phase. This requires a suitable weaving infrastructure that is able to apply aspects dynamically to already constructed applications. Especially a flexible application of aspects to a base while it is in its execution phase appears to be a highly attractive possibility. Use cases for this weaving strategy were already discussed in Chapter 1.5. The most important ones are the following:

- Rapid reaction to errors and security threads by fast roll-out of corrections: Aspects can apply quick corrections of the problem faster than a new iteration of the life-cycle might allow.
- Just-in-time customization of applications: This targets long-tail services that are derived from a generic base service application. Aspects can apply the needed modification that customize the application to the user's needs. This can be done just-in-time and automatically.

- Broad-scale roll-out of policy changes: Aspects can apply a new policy temporarily until a native implementation of them is done in a later cycle.

A common property of these use cases is that they all deal with changed or suddenly appearing new requirements and concerns while the application is deployed an in the execution phase. The common way of dealing with such changes would be to start a new iteration of the service application's life-cycle. This would mean to develop, construct, provision and deploy new versions of the services and business processes involved. This not only means that services in execution need to be interrupted and restarted, it furthermore might require a considerable lead time until a new version is available.

In many cases this full iteration through the life-cycle is not only acceptable, but actually necessary, because of the multiple technical, business and organizational dependencies associated with it. These dependencies are managed and controlled through the life-cycle model and business processes associated with life-cycle phases. However, these use-cases show situations where a different approach would be highly beneficial.

In particular two challenges are addressed. The first is a reduction in lead time until a change to an application is applied. This is in particular important for the correction of errors that would otherwise leave the services exposed to security threads, generate losses due to services not be bing available or gives users a bad usage experience. These situations can become highly critical for the business and demand immediate attention with suitable counteraction. The needed corrections are often just small changes in the application code. Aspects applied through online weaving could introduce these changes practically without delay and free of further impact on service availability.

The second challenge is the efficient implementation and roll-out of a new requirement to a large number of applications simultaneously. Policy changes might show this requirement, because policies become immediately applicable to the entire installed base of services. However, policy changes are usually known in advance allowing to prepare the service applications through a regular life-cycle iteration. Using Aspect Oriented Programming is therefore not vitally important, but it might lead to better planning and control of upgrades. Some services might be upgraded later when suitable while the policy change is already temporarily applied using aspects.

Individual customization of services is another use case where changes are applicable to a great number of services. It would ideally be done just-in-time in the moment a user requests the service. What customization is applied depends on the individual user that has requested the service. Especially in long-tail scenarios services are rarely used and only a small number of potential users ever actually requests a service that was particularly customized for them. In this scenario it does not appear to be particularly cost efficient to create every individualized variant of the service in advance, because many of these pre-produced variants would never actually be requested by a user. Using instantaneous just-in-time synthesis of the needed service variant is potentially much more cost efficient.

For this use case, Aspect Oriented Programming would be used to apply customizations to a generic base service application. This base service only contains the basic functions needed for every user. All individual customization options would then be added by means

of aspects just-in-time, when the service is actually requested. This example can even go as far as introducing a function that synthesizes customized advice code instantaneously with the service request. This means instead of many service variants an advice generator needs to be developed.

3.4.1 Dependency Between Aspects and Their Base Application

With respect to service life-cycle, the uses cases introduced in Chapter 3.4 have in common that aspects are treated as unique pieces of software containing the implementation of a particular concern. Furthermore, this notion of aspects includes, that they are applied as needed to a base application and not an integral part of it. With the presence of the aspect the base application gains additional capabilities or changes it's behavior. One variant of this concept would be a base application being a consistent piece of software even without the presence of any aspects. In general, at least certain aspects would be optional. In another variant the base application requires the aspects to create a consistent application, but there are a number of alternative aspects that can take this role of completing the application.

Treating aspects as unique pieces of software means that they have an independent life-cycle. Here the same life-cycle model that was introduced for SOA type services in Chapter 3.1 is also be used for individual aspects. In the analysis phase of the aspect life-cycle the concerns to be implemented are matched against the currently deployed base application in order to assess the feasibility of an aspect based implementation. This means the decision to use an AOP based implementation over other alternatives might be part of the analysis process.

In the design phase of the aspect life-cycle the point-cut criteria are specified. Furthermore the function to be implemented is broken down into advice units and inter-advice communication is specified. The following construction and test phase actually implements the advice and point-cuts based on the AOP infrastructure at hand. The actions done in the provisioning and deployment phases of the aspects are similar to those for regular SOA applications. The aspects are then rolled out into the execution infrastructure of the deployed base application for life weaving.

An aspect being in execution and monitoring phase means that it is weaved into all running instances of the targeted base application. This naturally demands that also the base application is in its execution life-cycle phase. In general, as long as the base application is in its execution phase the aspect life-cycle is relatively independent. It can, for example, go into a new cycle once requirements change. This leads to a new version of the aspect that is applied to the same version of the base application.

Aspects managed this way are only partly independent. They still depend on the specific implementation of a base application and the base application needs to be in execution, but as long as these conditions are fulfilled, the aspects have their own self-contained life-cycle.

3.4.2 Fragility of Point-Cuts

Fragility of point-cuts is a well known practical problem when AOP is used in software development [83, 84]. Aspects are usually developed with knowledge of the particular implementation of a base application. They are specifically made for this application. Only this way a detailed point-cuts can be written that captures unambiguously the exact set of locations in the source code, where changes need to be applied. A high attention to implementation detail is also needed when writing advice code that is able to apply all wanted modifications to an application's execution semantics and run time data. It is in this respect and in general not feasible to abstract and expose join-points through a generic API. This means that aspects are specific to a particular base application's implementation.

A problem arises when the base application's implementation changes, for example, if it is modified in order to implement additional features or for correcting errors. Modifications in the target code are likely to break the AOP implementation. The reason for this is that point-cut search and matching conditions might not find their target join-points in the code any more or they miss other join-points where advice would be needed. The result is that advice is not executed at the right join-points. Instead the point-cut might select unwanted join-points leading to erroneous advice invocations. But even if advice is executed at the right join-points, it might interfere with the new target code in an uncontrolled way. For example, data handling might have changed in the new version of the application, causing the advice to draw wrong conclusions when the new data is evaluated by the advice with the old logic.

It is possible, that the old advice implementations and point-cut definitions work well also with the modified base application. This would however not be a mandatory result of a controlled process, but happen as a lucky coincidence. In general, an application code change is likely to break the aspect implementation, and therefore the overall application.

A consequence of the fragility of point-cuts is that any change in the target code would require at least a review of the aspect implementation including the conditions for point-cuts. For the lice-cycle of an aspect with respect to it's base application this means that whenever the base application goes into a new iteration, the life-cycle of all related aspects need to iterate, too. This guideline implicitly ensures, that the aspects are re-considered as part of the development process whenever the base application is potentially changed. This is naturally a limiting factor in scenarios where aspects are developed and managed independently of a base application.

In the use cases of error correction and service individualization the base application stays stable within a life-cycle iteration of an aspect. Here aspects are particularly used in order to apply temporary changes and keep the entire application from going into a new life-cycle iteration.

Aspects and point-cuts can in principle be designed to be generally robust against changes in the base application This would however lead to severe design rules about the way aspects interact with a base application it might also limit the join-point model with respect to which join-points are available and what modifications are available to advice.

From base application point of view the following design rules would ensure that aspects

stay consistent and still apply the correct functionality:

- Join-points might only be removed or changed if it is certain that no aspect is currently using them.
- New join-points are only allowed if no aspect could erroneously use it.
- It is ok if existing point-cuts match against the new join-points, but the then weaved in advice must be needed at that location and interact correctly.
- Application data and state handling must be changed compatibly with the advice implementation and weaving rules.

These are severe conditions that, in general, cannot be met generically. In practice this usually means that all changes need to be applied considering the base application and all aspects together.

There is however the possibility to generate more robust point-cuts and aspects by designing and using a join-point model on a higher level of abstraction [83]. This is done by first of all introducing a conceptual model of the application. This model introduces a functional and structural abstraction and is then to some extent decoupled from implementation details. The join-point model would then be developed using the constructs used in the conceptual model. Using only this more abstract join-point model would make the respective aspects more independent of implementation changes. The lower level implementation might change, but as long as the conceptual model stays untouched, all aspects still fully apply. Fragility can then only be a problem if also the abstract conceptual model of the application changes.

The service selection constraints of the Ericsson Composition Engine constitute already a higher level of conceptual abstraction. It allows the developer to specify the requirements of a suitable constituent service rather than directly point at a particular one. This bears the potential to make aspect weaving more robust if the service selection constraint becomes part of the join-point model.

Weaving exclusively based on service selection constraints is however impractical as it only covers a part of the application logic. A useful join-point model for the Ericsson Composition Engine therefore needs to consider also less abstract elements of the composition definition, for example run time variables in the shared state. Thus, lower level fragility is still an issue. Nevertheless, service selection is the most important task in the composition logic. Consequently the abstract constraints that control the selection and respective join-points are practically the prime target of weaving conditions within a point-cut.

A particularly frequent use case of aspects is the replacement of one service in the composition by another one. This can be achieved entirely by weaving based on selection constraints where the constraints are also the only data that is modified. One of the most important use-cases is therefore directly more robust against implementation changes. This characteristic of the composition language of the Ericsson Composition Engine is a direct consequence of the availability of higher abstraction in design. This is a general finding: the use of higher abstraction and model based design will lead to more robust aspects.

3.4.3 Aspects for Multiple Targets

In the use case of error correction and service individualization the aspect is designed particularly for a single target application. In contrast, the broad roll-out of new policies to many services would mean that the same aspect might be applicable to many service applications. This means that new functionality is rolled out broadly to many service applications through weaving of a single aspect. The point-cutting conditions of the aspect need to be written so that they are applicable to multiple target base applications at once.

A first technical prerequisite for this scenario is that the weaving infrastructure must support a single aspect targeting multiple base applications. This is the case, for example, for the AOP implementation of the Ericsson Composition Engine that was developed as part of this thesis. All composite applications deployed in and executed by the Ericsson Composition Engine are by default target of all active weaving instructions. The weaving condition is therefore by default checked against all applications. In this respect, limiting the weaving to a single application would be an additional condition in the weaving rules.

The technical challenges for writing aspects that can ba applied to multiple different target applications are similar to the challenges imposed by fragility of point-cuts. This is not a trivial task. The first big challenge would be to develop advice code that is able to operate within the implementations of multiple target applications. This requires a certain level of similarity on the base applications' implementations. The advice code might, for example, need to evaluate run time data that is ideally represented equally in all targeted applications. If this is not the case or if the differences cannot simply be resolved in the advice, the use of aspects to broadly distribute a feature is questionable due to the complexity of advice design.

The related second challenge would be point-cuts that correctly weave advice across applications. Also here, similarities in the join-points are required and weaving rules need to focus on them.

These are severe challenges that limit the cases in which aspects for deployment to multiple applications are feasible. In practice, when an entire portfolio of applications is build form the same constant toolbox of components, the needed similarities can be found and aspects can consistently target them across applications. Also here, application design based on a conceptual model helps as its abstraction further limits the diversity in potential weaving targets and advice based modifications.

These considerations lead to the conclusion, that Aspect Oriented Programming can benefit from higher abstraction in application design. It can enable the use of AOP in cases which would usually be to complex. Languages for defining service composition operate already on a fairly high level ob abstraction. This is the case for BPEL and BPMN, but even more so for the Ericsson Composition Engine with it's constraint driven service selection mechanism. The conclusion is that AOP and service composition are actually a very good match.

3.5 Dynamic Management of Concerns and Aspects

An overall consistent application usually needs to address a number of concerns. Next to a core function delivered to the customers, it implements a couple of additional concerns, for example charging for usage, measurement and control of service quality, or general logging of all user actions. If AOP is available, some of the concerns are still addressed directly and natively by the base application, while others might be added by means of aspects. Management of aspects basically refers to management these concerns and their implementation with respect to the questions, which aspects shall be applied and when they need to be active.

Dynamic aspects refer to an environment where aspects have a partly independent life-cycle as discussed in Chapter 3.4.1 and where they are entities, which can be developed and managed independent from a target application or from each other. It is in principle possible to even find a different aspect configuration for every run time instance of an application. Especially the use of online weaving provides this ability.

Basic Aspect Deployment Control and Execution Tracing

In the solution for AOP, as proposed in this dissertation, aspects are in general not exclusively applied to single applications, but rather globally to the execution environment. If an aspect shall only be applicable to a particular application, this needs to be explicitly asserted within the weaving rules. In general, rules, as expressed in weaving instructions, determine dynamically, if and which aspects are applied. This decision is taken dynamically at run time for every executed application session. While this approach enables a high degree of flexibility and dynamic behavior, this same behavior can become a problem fo the developer. It can lead to extremely complex dependencies, while the developer needs to guarantee correct behavior for the overall applications and correct dynamic deployment of aspects.

A minimum requirement is therefore the availability of suitable tool support, which provides the following basic management functions:

- Management of Weaving Rules: All weaving instructions being active in the service execution environment are visible to the developer. Furthermore they can be added, modified and removed.
- Management of Advice: The deployed advice is presented to the developer. Advice can be added, modified and removed.
- Tracing of Advice Weaving: Executed applications can be traced with all constituent services and full state data context. This needs to include the weaving rules being checked and advice being executed.

These tools would allow a developer not only to setup aspect weaving, but also to trace and debug the full execution history within its original data context. The Ericsson Composition Engine is paired with an integrated environment with tools for composition design,

service description management and execution tracing. This already existing collection of tools was extended in order to support the basic aspect management features as described in this chapter. Thus, the baseline tool support needed was demonstrated as part of the proof-of-concept implementation.

Rules for Global Aspect Management

Aspect management does not only refer to the ability of targeting an application flexibly with a particular set of aspects. An important challenge is also the management of concerns, which shall or shall not be added to an application. This translates into managing which application shall get a function or characteristic and when this new property shall be added.

Weaving instructions, as developed in this dissertation, provide a basic first layer of these rules. They allow, for example, to explicitly refer to the name of a composite application. These rules are however based on the join-point model and the shared state data context of the composition execution session. This means that only application internal data can be checked against these rules. Global considerations and contexts, which are not reflected in the application data, are beyond reach for the aspect deployment logic on this level.

Weaving rules can however be modified dynamically through the management API of the composition execution engine. Loading and removal of weaving instructions directly enables or disables aspects. Or they at least partially change their weaving behavior. Consequently, another layer of rules can be introduced. It would utilize the basic weaving instruction management interface for manipulating the deployment state of aspects. Well known and widely used rules engines, such as DRools [85], appear adequate for the task. They apply rules with a basic event-condition-action scheme. Furthermore, it is possible to integrate the rules with a great variety of input data sources, such as databases or event management systems.

A practical example for these rules would be the implementation of a general emergency mode for all applications. In this mode, some services change their behavior in order to not grant user access to security critical features. For example, a banking website might still allow the users to see their account status, but transactions cannot be initiated. This emergency mode is ideally only applied temporarily and in rare occasions. Applying it dynamically by means of aspects keeps the applications free of the respective logic until it is really needed. If it is needed however, a rapid system-wide deployment would be required. Furthermore, also already executed applications need to be changed into emergency mode operation.

The weaving rules overlay can be deployed in order to reach this behavior. A set of rules is designed, which waits for a certain condition, such as a security alarm, to apply. The action associated with these rules, would momentarily add a couple of additional weaving instructions to the composition execution environment. The respective advice would already be deployed, but without the related weaving instructions it would be passive code. In the emergency mode it is active and changes the behavior of all services momentarily.

If the security alarm is ceased, another set of rules can remove the weaving, thus, allowing normal operation without the emergency aspect.

This aspect management mechanism uses a security alarm event. This is data, which is typically available within the OSS infrastructure but not in application execution sessions. This means it would be beyond reach of an implementation within individual applications in their usual execution environment, regardless if it is done using AOP or not.

Further refinements of this example are possible: For example, the emergency mode can be limited to a particular but dynamically selected group of users. Defining this group is another example of external global data, which the rules need to consider. Here, it might be beneficial if the rules can not only apply predefined sets of weaving instructions, but if the rule's action can parameterize a weaving instruction. This means the weaving instructions would be re-written dynamically before it is deployed. In this example an additional condition based on identification of effected users would be added to the weaving instruction. The related aspects would still be deployed globally, but the user specific condition would need to match against user information in the application run time sessions. Thus, emergency mode advice is only selectively invoked.

Management of Aspect and Concern Interaction

The decision to add a new property to an application or an instance of it can depend on any type of context or situation. An important factor is, for example, the user, to whom the application provides a service. For example, any use case involving individualized and customized applications, needs to take ad-hoc decisions based on the requesting user. But any other kind of context, such as current time and date, location of the user or recent performance shortage in the network, might also be relevant. In general, a flexible set of rules might be deployed, as described in Chapters 3.5 and 3.5, in order to decide automatically about adding an aspect to an application instance or removing it when not needed any more.

In a flexible environment like this, it is essential to ensure that each concern is only addressed once within an application instance. This condition includes the base application and all added aspects. It must, for instance, not happen that aspects implement and add a function, which was already addressed by the base application or added by another aspect. This can easily happen if two units of a service provider independently develop aspects and contribute rules for adding them. The consequences can be as severe as double billing if, for example, a charging concern is addressed multiple times.

This chapter briefly outlines a method that allows automated management of concerns. It stops the weaving of an aspect, if its concern was already addressed by another aspect or by the base application itself. This function is subject of the patent application [86]. It was developed as part of this thesis work.

The method described in the patent [86] keeps track of all concerns provided by an application. It introduces a concern manager as additional function included in the weaving and execution engine. It implements bookkeeping of all concerns. This includes the concerns already available in the base application and also those, which are dynamically

added by aspects. The underlying weaving engine would only add aspects to an application instance if approved and ordered by the concern manager. It would in particular not allow advice to be invoked, if it implements a concern, which is already registered for the application instance. Thus, it actively avoids that a single concern is addressed multiple times.

The concern manager, together with global weaving rules, constitute a meta AOP environment. It analyses the entire set of available applications and concern. This might lead to automatically applying a set of additional aspects for adding required but not yet addressed concerns. The rules for this are another level of point-cut, which does not only target a single application specifically, but rather the entire service environment of the service provider across all applications and aspects.

The prerequisite for this concept is an extended service description for all applications. Furthermore, also aspects are described similarly. This service/aspect description contains a list of tags that denote all individual concerns addressed by an application or an aspect. If the application is composite, then its addressed concerns are determined indirectly by the set of all used constituent services. Each application needs to be described this way.

Also the aspect is described with a similar list of addressed concerns. This can be additive, meaning that the advice will add a concern to the base application. The weaving can on the other hand also remove the implementation of a concern, and thus, remove the concern. In any case the concern manager will keep track of all changes to the concerns configuration as implied by service composition and aspect weaving. When the execution of an instance of the composite application starts, the list of addressed concerns is loaded into a concern manager. Other than this the composite service execution is started as usual together with online weaving.

It is however possible, that the AOP based implementation of a concern is spread over several individual advices. They all cooperate in order to jointly address the same concern. It must therefore not happen that the first advice from this set is executed causing a blocking of all further weaving of the same advice or of the other associated advices. For this reason, each dependent set of advices shares a unique ID. The concern manager uses this ID and maintains not only the list of available concerns, but it also notices, which advice or application has contributed or removed a particular concern. If within this execution session another advice identifies itself with an already known ID, its weaving is granted.

With respect to the outlined method for concern management, there are two ways a composite application can change dynamically the set of considered concerns. The first is by selecting a particular constituent service that addresses the concern. The second method is through aspect weaving. It is in this respect interesting that aspect weaving and constraint based service selection are two different but complementary mechanisms for addressing concerns within composite services.

Chapter 4

Concept Development

Chapter 2.2 has introduced a highly flexible service composition mechanism that was designed specifically for challenges and requirements in the telecommunication domain. Based on this foundation this thesis will construct a solution for Aspect Oriented Programming that can help solving important business needs:

- Enhanced reusability due to optimized separation of concerns, and in particular of cross-cutting functional concerns.
- Availability of an efficient mechanism for creating customized service application variants.
- Low operational expenses through reduced composition complexity.
- Fast implementation, roll-out and management of global policy changes to the entire installed base of service applications.

This chapter analyzes these requirements and presents the steps of developing a solution.

4.1 Challenges to be Addressed

4.1.1 Composite Service Complexity

Any application, and thus, also those implemented by means of service composition, need to address a number of concerns. The starting point is usually an idea for a new application. Its basic functionality would be comparably lean if the only concern would be to provide the core function to the user by implementing the initial idea.

In practice there are many additional concerns that need to be addressed and that do not directly contribute to providing the actual function of the application to the user. What might happen to a composition skeleton if such additional requirements are implemented within a composition skeleton is shown in Figure 4.1. The skeleton on the left is focused

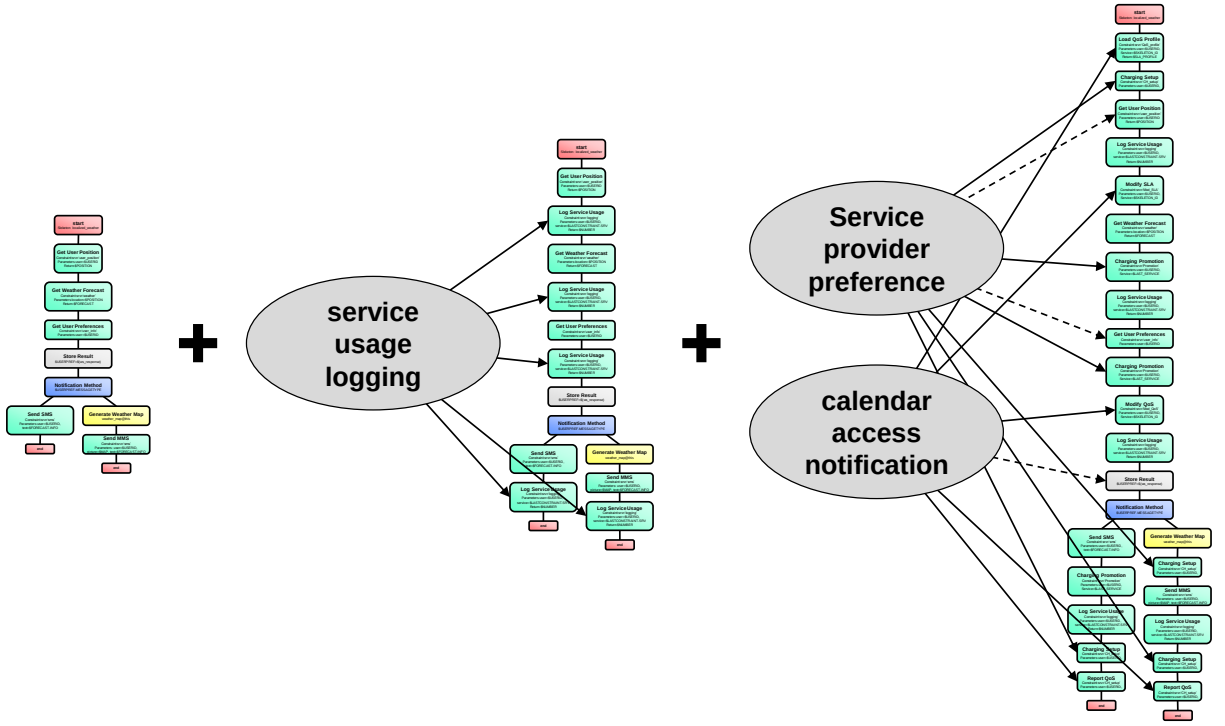

Figure 4.1: Increase of implementation complexity when implementing additional concerns

on providing an end-user service only. It is small and only contains elements necessary for the core business logic. in addition, the operator might, for example, require logging of all service invocations. Logging is a typical simple network operator requirement. Implementing this obviously cross-cutting concern by means of skeleton design is relatively easy. It can, for example, be achieved by adding an invocation of a logging service after each of the original service templates.

While the skeleton is still reasonably lean after adding the logging concern, its size and complexity might explode if further concerns are addressed in a similar same way. Figure 4.1 also shows the implementation of two more concerns. One is a policy that limits service selection to certain providers and the other is a function that notifies a user if other users access his calendar. Already after adding three simple additional features, the former lean skeleton became complex. This example only shows the addition of service templates without also applying major structural changes in the skeleton, such as adding further conditional branches. After further addition of a couple of more concerns, the skeleton can easily become hard to understand and to maintain due to exploding complexity. Furthermore, the resulting application gets highly specialized. It is therefore likely that it cannot serve as a suitable constituent service in other compositions.

This example demonstrates the extent to which also service composition suffers from cross-cutting of concerns. Skeleton based composition, although it can be highly dynamic, is still based in basic functional decomposition, and therefore fails to solve cross-cutting concerns without considerable complexity.

4.1.2 Policy Implementation

Policies express general guidelines for services. Concerns addressed by policies are often strategic or driven by general business or marketing considerations. For example, services would be required to comply to a certain charging and billing interface design, or some providers of constituent services could be blacklisted and must not be used while others might be explicitly preferred. Also legal requirements and public regulation are a common source of global policies.

Usually all services offered by a service provider would need to comply to the policy. Consequently a policy constitutes a cross-cutting concern. It's implementation would not only cross-cut a single application, but the entire service domain with all deployed services.

4.1.3 Service Customization

The long tail market is characterized by services that have only a small number of potential customers. Addressing this market requires the ability to create these services with high cost efficiency. A good way of doing so would be the creation of customized variants of existing services. Using AOP, these custom variations of a service could be implemented.

4.2 Solution Overview

In order to create an AOP solution, a couple of basic decisions need to be taken:

The weaving mechanism: There are two fundamentally different ways of applying aspects: offline weaving and online weaving. As outlined in Chapter 4.3 online weaving has advantages if run time data and data driven decisions are central for the composition mechanism. This is the case for the constraint based service selection mechanism of the Ericsson Composition Engine. For this reason the AOP solution is developed based on online weaving.

The join-point model: A join-point model is needed. This addresses the basic question of which patterns in the target language can be captured by the aspect weaver in order to apply an advice. Here the composition language of the Ericsson Composition Engine is the chosen target language. The dynamic and constraint driven service selection at run time is a property of particular interest with respect to AOP. Join-points are defined based on the composition language. Chapter 4.4 explains the developed join point model in detail.

Specification of weaving: A weaving engine needs instructions that describe which advice shall be applied to which join-points and under which conditions the advice shall be applied. A language that allows expressing these weaving instructions is therefore another key component of an AOP solution. The proposed weaving language is introduced in Chapter 4.5.

Specification, invocation and execution of advice: Advice is implemented using the same programming language as the targeted base language. This means advice is implemented as additional skeletons. It can therefore be executed directly by the composition engine. The details of advice selection and invocation are explained in Chapter 4.7

Data exposure: Composition sessions handle data. In this context the relevant data is the session data for managing the composition session and data used by the implemented composite application. Protocol data and, in general, the entire shared state are examples of the latter. It is essential to define how data is exposed to AOP specific functions, what data is available to the evaluation of weaving instructions and how the advice might apply changes. Chapter 4.6 explains the proposed solution.

4.3 Choosing the Weaving Paradigm

The choice of the weaving paradigm addresses the decision, if dynamic or static weaving shall be used. Both have specific advantages and disadvantages.

Service composition as done by the Ericsson Composition Engine has to take a decision which service to select in order to instantiate a service template. The default decision strategy is purely based on constraints expressing mainly functional criteria. If several constituent services are possible, the first in the list provided by the service database is used. There might be the need to further optimize the service selection. One example is load balancing. The selection strategy might need to evenly distribute the number of invocations across all available services of the same type. Or the service shall be used, that is provided by the server with lowest load. If the constituent services are provided by 3rd parties for a fee, a good strategy would be to prefer the cheapest available service. The strategy to be applied might depend to great extent on run time data, for example the user's subscription. Thus, it is not possible to select the right strategy at design time. Dynamic AOP based on online weaving would be able to take a weaving decision based on the run time data and apply the right weaving strategy.

Offline weaving applies advice at design time by creating a consistent new application that is addressing all concerns. The weaving is therefore part of the build or deploy process. It allows to apply advice based on all data that is available offline. This is basically the skeleton itself with its structure and everything that is specified in the skeleton elements. The weaver could, for example, automatically add services to or remove services from the composition. It can modify constraints and branching conditions and, in general, it can change everything that is specified directly by skeleton elements.

Many useful modifications can be implemented based on offline weaving. Even a dynamic reaction on data that is only available at run time can be achieved by writing aspects that contain checks based on the run time data in order to determine what to do. Aspects need to be weaved into the target application in all locations where potentially a functional change is needed.

However, offline weaving cannot apply aspects conditionally based on data that is only available at run time. This would be case for the entire shared state data, or the selected and instantiated constituent services. When using offline weaving, many checks can be integrated into the weaving instructions and the advice is only added and executed if its need is implied by the run time data. The advice code can therefore be expected to be smaller and less complex when written for online weaving. A side effect of this is also a

better re-usability of the advice.

A solution for offline weaving can be implemented similar to the proposal in [75]. Also skeletons used in the Ericsson Composition Engine are represented as XML. Thus, an XPath based point-cut can easily be applied.

Online weaving has also disadvantages. Most importantly the process of online weaving uses capacity of the application server in productive operation. Capacity is a precious resource because it directly influences important key performance indicators of the server, for example the number of users that can be served per second. Thus, AOP with online weaving has CAPEX disadvantage.

The decision to choose online or offline weaving therefore depends mainly on the extend to which weaving and advice functionality will be based on run time data. If in practical use cases run time data has high importance for the aspect weaving and the advice functionality, online-weaving appears to be the best choice as long as the needed capacity budget stays acceptably low.

The decision which constituent services shall be used and the control of their execution is the single most important function of service composition. With the Ericsson Composition Engine the entire selection of a constituent service is data driven and performed at run time. This run time mechanism is consequently expected to be a main target of aspects. Consequently in this composition environment, the run time data will be central for developing advice and for the specification of weaving. This results in the decision to use online weaving in the Ericsson Composition Environment. This preference for online weaving also holds for all composition environments with similar characteristics as long as the capacity loss stays within acceptable boundaries.

This thesis puts special attention on the application domain of telecommunication services with their strict requirements on low latency and real-time responsiveness. Therefore, the application server's Load characteristics are highly important for meeting the required levels. The validation of the capacity loss due to the presence and operation of the online weaver is therefore a key aspect of Chapter 5.

4.4 Defining a Join-Point Model

When specifying an AOP framework one of the main tasks and usually the starting point is the definition of a join-point model. The weaving engine is created based on this model. Join-points are those locations in the targeted programming language where advice can be applied. A join-point model therefore defines what elements of the targeted programming language constitute join-points. In this chapter the join-point model for a heterogeneous service composition of the Ericsson Composition Engine is defined.

In all cases, were an existing programming language is extended with Aspect Oriented Programing capabilities, elements in the programming language are identified that will be used as join-points. These are language elements that are most essential to the application logic, and therefore also the most essential targets for changes.

The target language here is the skeleton language of the Ericsson Composition Engine.

The basic idea is to define a join-point in each of the skeleton elements. This leads to a lean join-point model that would consist of only six join-points corresponding to the start, end, service template, SSM command, condition and goto skeleton elements.

Each of the skeleton elements represents a complex activity rather than an atomic operation in the sense that it constitutes a process of several well defined actions within the underlying composition execution engine. Thus, it is not sufficient to just identify the element. The weaving engine needs more specific indications when the advice shall be added and executed. It is necessary to identify specifically when in the context of a skeleton element execution advice shall be applied.

In order to enable a more specific control, the join-point is accompanied with the specification of weaving control hints. This thesis proposes the introduction of the execution hints BEFORE, AFTER and AROUND.

BFORE: Advice is executed before the activity related to the skeleton element that constitutes the join-point.

AFTER: Respectively, first the join-point action is execute and then advice is applied.

AROUND: The AROUND weaving control hint leads to skipping the execution of the join point skeleton element. The advice is executed instead. The AROUND constitutes a replacement and if used together with an advice that does not perform any action, it can be used to delete skeleton elements.

Which way of advice execution is chosen depends to a great extent on the actions the advice is supposed to do. If the advice changes data that is input to the respective join-point skeleton element, it needs to be executed before the skeleton element. If the advice reviews the decisions taken or the results created by the join-point skeleton element, the join-point element needs to be executed first.

It is also required that an advice leaves the composition in a defined and valid state that will not break the skeleton execution. Especially the removal of certain elements might be problematic in this respect. For example, removing a condition element would leave the composition execution in a state where it is not clear in which branch the execution shall continue. The advice is of course allowed to overrule any decisions taken by the base skeleton, but it is obliged to apply the changes compatibly and consistently.

4.4.1 Start of Skeleton Execution

This join-point captures the start of skeleton execution and corresponds to the skeleton start element. The following join-point specific actions can be done by the advice depending on the weaving control hint:

BEFORE: The skeleton start element marks the start of the composition execution for a given skeleton. The question is what actions any advice can do before the composition has actually started. There is no shared state yet, because it is created when executing

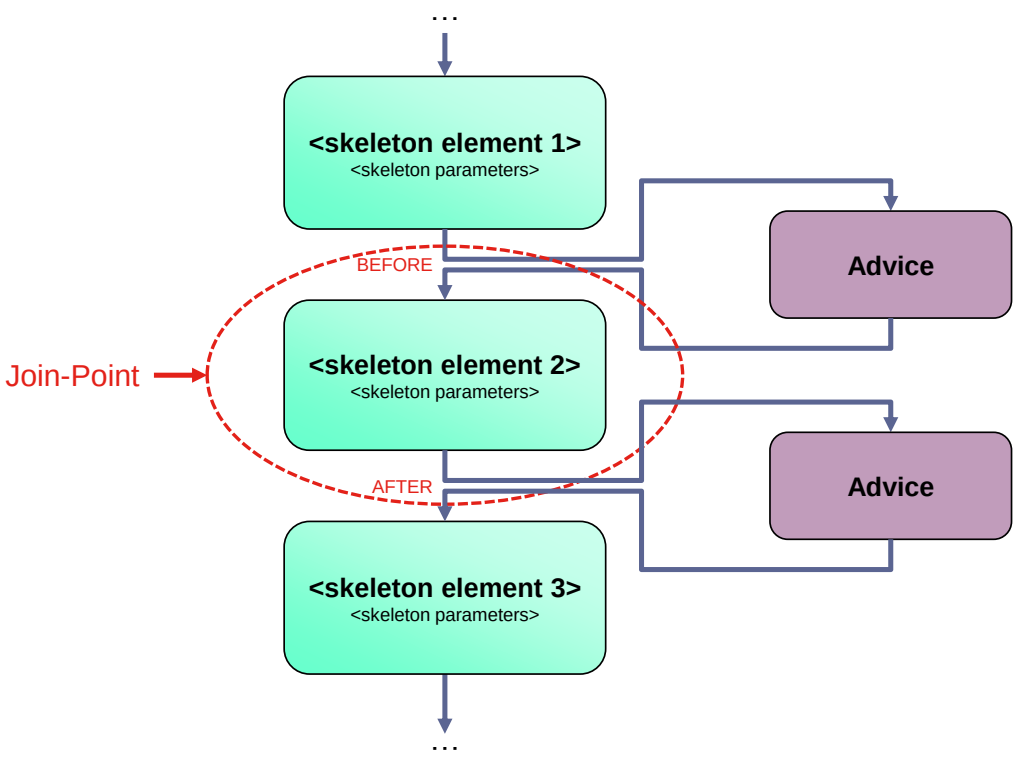

Figure 4.2: Skeleton element as join-point and advice weaving for the BEFORE and AFTER weaving control hints

the start element itself. As the shared state is a prerequisite for basically all actions of a skeleton, there is no sensible action an advice can do. Theoretically, it would be possible to create a composition engine implementation that actually allows advice actions even before the start element. An example would be the execution of service templates that invoke additional services. However, in practice there is hardly an action that really need to be executed before the start element. Executing it as the first element after the start element is usually sufficient. Actions weaved in by advices directly after the start element would still be the first actions performed in the composite application. Nevertheless, the start element has parameters. They influence the selection and start of the skeleton. Once the skeleton is already in execution, these parameters do not have any influence any more. As the AOP weaver is closely related to the skeleton execution, no advice can have outreach to and an effect on decisions taken before the skeleton execution was started. This is simply a matter of causality. Thus, the BEFORE weaving control hint for start elements is not applicable and stays without implementation.

AFTER: As mentioned above, advices weaved into the composition after the skeleton start element would define the first actions of the composite application.

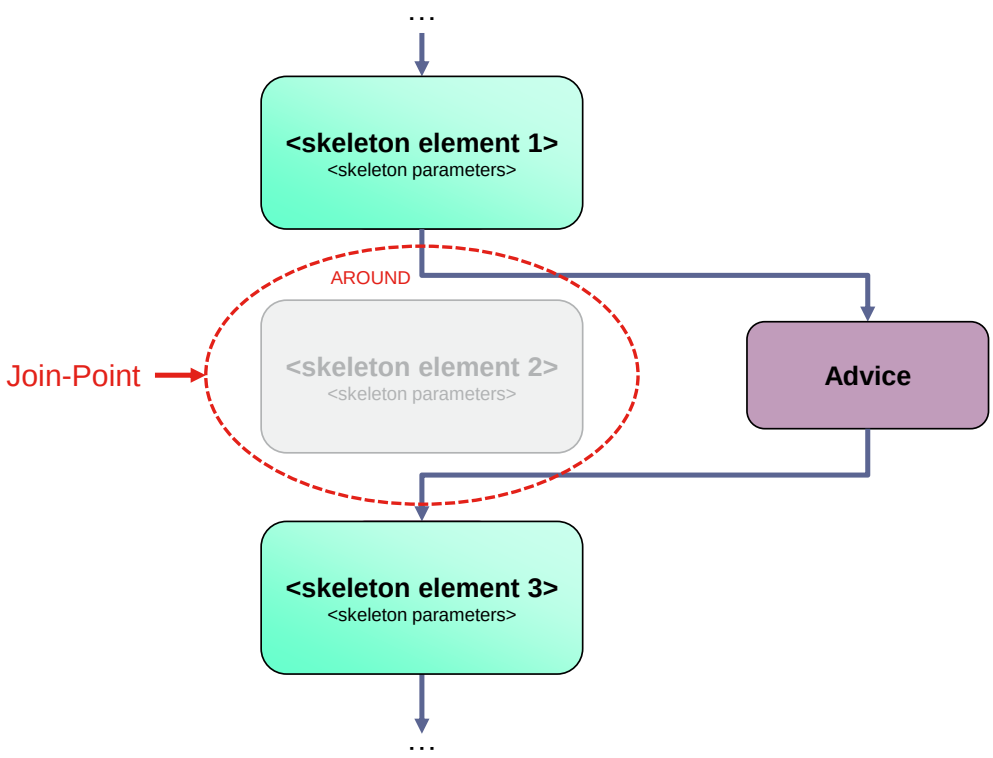

Figure 4.3: Skeleton element as join-point and advice weaving for the AROUND weaving control hint

AROUND: Also the AROUND weaving control hint is not applicable for start elements. A skeleton needs a starting point of execution. Thus, a start element is not optional and must not be removed or replaced by something else.

4.4.2 End of Skeleton Execution

This join-point corresponds to an end element and it therefore marks the termination of a composite application.

BEFORE: Weaving an advice into the composition before an end element means that the actions implemented by the advice are the last ones of the composite application.

AFTER, AROUND: AFTER and AROUND weaving control hints are not applicable for the end element. The argumentation is similar to the one of the BEFORE and AROUND of start elements. The composition session is terminated by the execution of the end element, thus, all basis for further action in this composite application is not available any more. Additionally all advice that needs to be the last action of the application, can be weaved in right before the end element.

4.4.3 Service Template Join-Point

The central task of a skeleton and the composition engine is providing a composite application by selection and execution of constituent services. Consequently, the join-point related to the service template is the most interesting element for aspect based modifications, and thus, also for dynamic weaving of advice.

For the service template join-point all three weaving control hints are available:

BEFORE: If the advice is executed before the service template, changes to the parameters of the service template would be considered later when this join-point's service template is executed. If, for example, the service selection constraints are changed, this will effect the service selection query, thus, a different constituent service might be selected in order to instantiate the service template. Also changes to the service parameters would have an immediate effect on the invocation of the constituent service.

AFTER: Execution of advice after the service template join-point means that the result of the constituent service execution is available and can be reviewed and acted upon. This includes, that the result reported by the constituent service might even be modified before the skeleton execution resumes after the advice execution.

AROUND: The around execution of the service template allows to replace the entire service template. In this case the service selection and service invocation join-points of this service template might never be reached, because only the advice is executed.

4.4.4 Service Selection and Service Invocation Join-Points

The service template element combines two basic processes of the composition execution engine: service selection and service invocation. The service selection process chooses the constituent service to be executed. It uses the service selection constraint parameter to formulate and send a query for constituent service candidates to the service database. The result of this action is a list of service candidates that fulfill the selection constraint. All services in this list are suitable for instantiating the service template.

The service invocation process is the execution control of the one constituent service that is chosen from the list of service candidates. By default simply the first service candidate from the list is chosen. In order to perform the constituent service invocation, the service parameters from the service template are used to build the respective request message to be sent to the service. The reply of the service is written into the shared state according to the return parameter that is also specified in the service template.

From pure composition execution point of view service selection and service invocation are always performed together. With respect to AOP the internal processes of the service template might individually be subject to review and modification. For example, the default method of selecting simply the first service from the list of service candidates might be changed. The advice might, for example, introduce further criteria for selection in order to enforce a particular candidate service .

The already introduced service template join-point would not allow an advice to apply these modifications. The reason is the order in which results are generated within the execution of the service template sub-processes. If the advice would be weaved in before the service template, service selection was not yet performed. Therefore, the service candidate list is not yet created and cannot be inspected and modified. Advice being executed after the service template would be able to access the candidate list of constituent services, but any modifications would be pointless, because one candidate service was already selected and executed. Consequently, an advice that needs to interfere with the service that is selected and executed must be weaved in-between the selection and the invocation sub-processes of the service template.

For this reason, two additional join-points are introduced: Service Selection Join-Point and Service Invocation Join-Point. This splits the service template skeleton element into its two sub-processes within the join-point model. These additional join points are nested within the service template join point. The execution order of advice with respect to using BEFORE and AFTER weaving control for Service Template, Service Selection and Service Invocation Join-Points is shown in Figure 4.4. Also AROUND weaving control is available for all three join-points of the service template individually. If however AROUND is used with the service template join-point, the entire service template might be replaced. Thus, service selection and service invocation join-points would not be reached.

Service Selection Join-Point

The Service Selection Join-Point is first join-point that is reached within the execution of the Service Template skeleton element. Aspects using this join-point can directly interfere in the service selection process performed based on the service template parameters.

BEFORE: An advice selected with a BEFORE weaving control hint is executed after the BEGIN advices of the service template join-point, but still before the service selection process is assembling and sending the selection request to the service database. The advice can change the service selection constraint, thus, directly influence the service selection. The join-point is executed within the service template context. This means also the other parameters of the service template can be changed, although they are not used in service selection but later in service invocation.

AFTER: When the advice after the service selection is executed, the service database has already provided a list suitable service candidates and the first service from this list is scheduled for invocation. An advice can review and modify the list, thus, change the service that is actually invoked. This can, for example, be reached by re-ordering the candidates list, but also completely new list entries can be added ot the entire list can be replaced. Also here, the service template parameters that are used later in service invocation can be modified.

AROUND: The advice execution around the service selection replaces the service selection process. The result is that no service is selected. This would lead to no constituent

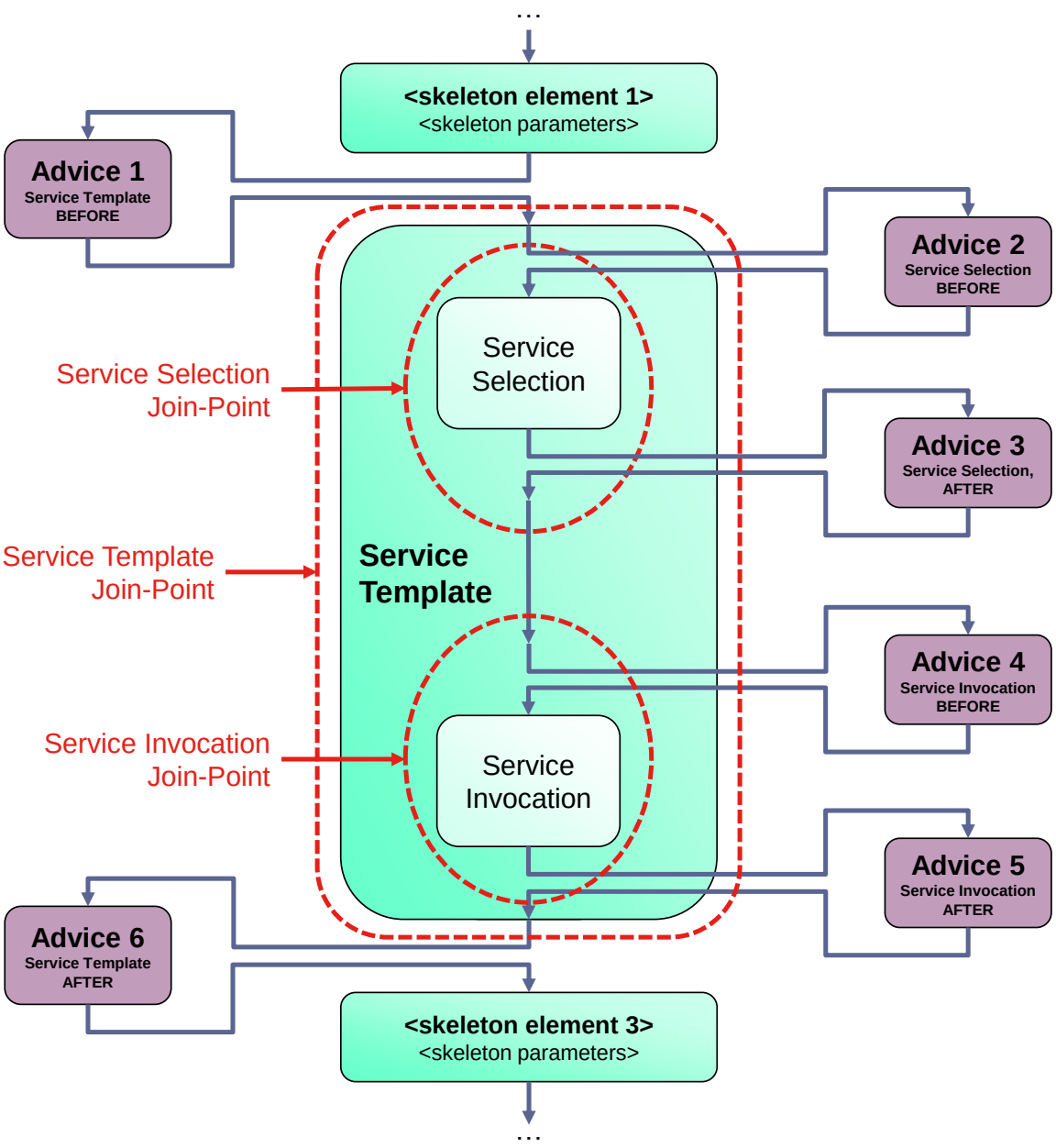

Figure 4.4: Join-points and advice weaving at the service template

service being executed, unless the advice creates a list of service candidates itself. The advice would replace the request to the service database and service invocation has still all input it needs. The advice might achieve this by performing the service database query itself or it might implement a different logic for finding and selecting service candidates.

Service Invocation Join Point

This join-point marks the execution of a previously selected service candidate. It therefore catches the second basic sub-process within the execution of a service template.

BEFORE: Before the Service Invocation Join-Point, an advice can change the service parameters and the variable that later receives the result. Furthermore the advice can still modify the service candidate that is scheduled for invocation. From advice point of view, using the BEFORE weaving control hint of the Service Invocation Join-Point is similar to using AFTER weaving control hints at the Service Selection Join-Point. The same run time data is available at both join-points and also the same modifications are possible. The reason that both exist is mainly conceptual consistency of the join-point model.

AFTER: Advices with AFTER weaving control hints at the Service Invocation Join-Point are executed after the selected constituent service has finished execution and before the advices at the AFTER weaving control hint of the service template. At this point return values of the executed constituent service are available. One possible advice action is evaluation and modification of the returned service result values.

AROUND: Advice executed around the service invocation avoids that the selected service is actually executed. Instead, the advice can substitute the service execution with other actions. This might be based on the previously selected service candidates, but the advice can also completely ignore what the service template specifies and execute something entirely different. The result variable can be written by the advice before the skeleton execution resumes.

4.4.5 SSM Command Join-Point

The SSM command skeleton element allows value assignments to shared state variables. The only parameter of the SSM command skeleton element is the command expression.

BEFORE: The SSM command expression can be changed by the advice enforcing a different shared state variable operation to be executed.

AFTER: The advice can use another SSM command in order to write into the same variable as the original SSM command of the join-point. Because the original SSM command was executed earlier, the changes applied by the advice persists.

AROUND: The advice replaces the SSM command. The SSM command expression can also be cleared with an advice weaved in before the join-point, and as a result, the SSM

command would not have any effect. Thus, AROUND is in principle not needed. it is proposed only for conceptual consistency and completeness

4.4.6 Condition Element Join-Point

While the service templates mainly contributes to the set of constituent services to be used, the control-flow of the skeleton determines the execution order. The condition element designates alternative control flows by conditionally branching into different parts of the skeleton. The condition element is therefore an important target for aspect weaving and modification, because it allows advice to directly impact the control flow.

At the condition element an expression is evaluated that formulates a branching condition usually based on shared state variables and run time data. The evaluation result determines in which out of several skeleton branches the execution continues.

BEFORE: Advice can modify the expression of the condition element. Consequently another branch of the skeleton might be taken.

AFTER: The after weaving point for the control element is the first action within the chosen branch. The branch label would be available to indicate to the advice in which branch it is executed. There are not many uses of the weaving point as it was therefore not implemented.

AROUND: Around weaving at the condition element is facing a basic problem. It is supposed to skip the condition element, but there are several possible branches for resuming after the advice has finished. In which of them shall the execution continue if the advice has skipped the condition evaluation? A solution might be to determine the wanted branch from within the advice. This would be mandatory, because a default branch is not defined. In practice this would not really remove the condition element's function of selecting a branch. The advice would be forced to do so instead, but the branch selection as such stays. The same behavior can also be reached using an advice at the BEFORE weaving point modifying the condition expression. All the effort to create also an AROUND weaving point would not create significantly new possibilities. Consequently, it is not implemented.

4.4.7 Goto Join-Point

The Goto Join-Point allows to influence the call of a sub-skeleton or the jumping within same skeleton execution.

BEFORE: Advice can modify the destination specifier of the goto element. As usual, it can point to a sub-skeleton or to another element within the same skeleton. The goto element can also specify parameters. They are basically shared state variable assignments similar to SSM commands. Nevertheless, they are also subject to modification by advice.

AFTER: Advice can also be weaved in after the goto element. For this point, there are no modifications specific to the skeleton element available, but additional actions can be introduced.

AROUND: Advice executed around the goto skeleton element removes or replaces jumping or the call od a subordinate skeleton. Instead only the advice is executed.

When using a goto element within an advice, it is only possible to call other skeletons, or to jump within the same advice skeleton. In the reference implementation jumping back to the parent skeleton from an advice execution is not possible by means of a goto element. The advice execution needs to terminate using an end element in order to go back to the parent.

4.5 Weaving Language

Online weaving relies on a run time weaving function that first of all detects if a join-point is reached while executing the skeleton. For this purpose a weaving engine is implemented into the composition execution engine. At run time of a composite application the weaving engine monitors the respective skeleton execution session. If a join-point is reached, the weaving engine identifies if advice needs to be applied and which advice to select. Finally, it initiates and controls advice execution.

Weaving instructions define the basic rules that guide the weaving engine in this process. This Chapter explains how weaving instructions are expressed by using a weaving language. The newly developed weaving language is described. It directly corresponds to the chosen join-point model as introduced in Chapter 4.4.

A suitable weaving language needs to enable a developer to specify:

- The join-point at which weaving shall be performed,
- A condition for weaving based on the execution context,
- The exact weaving point by providing a weaving control hint,
- The advice that shall be weaved in and executed.

4.5.1 Composition Execution with Weaving

The online weaving engine is a fully integrated part of the composition execution engine that interprets skeletons in order to execute a composite application. The actions defined in a skeleton directly translate into actions of the composition execution. Which actions of composition execution are picked in weaving, is determined by the join-point model. In this respect the weaving execution engine is similar to a rules engine where the weaving instructions are rules and the skeleton execution run time creates a stream of data being

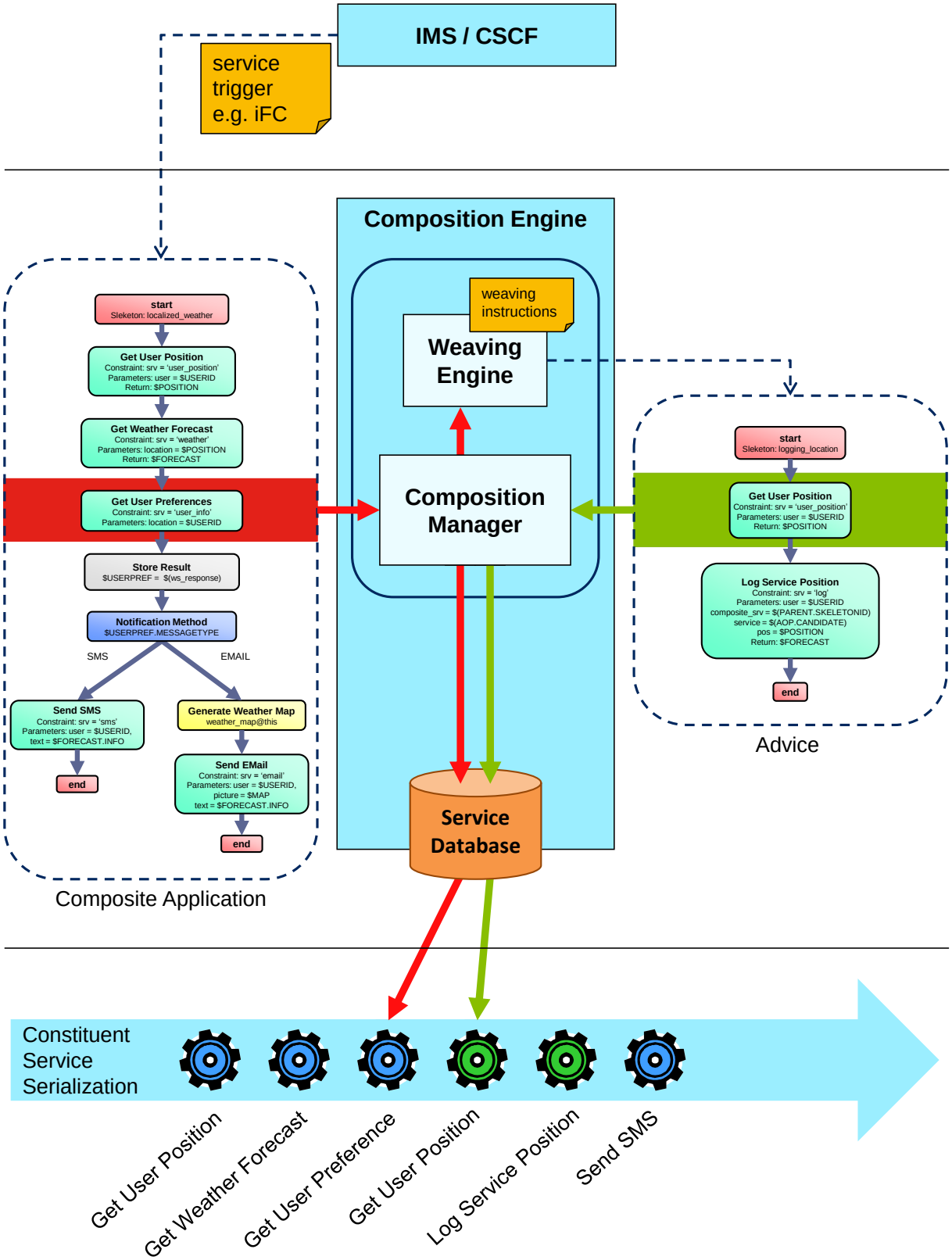

Figure 4.5: Execution of a composite application with advice weaving

checked by these rules. They determine, at which observed pattern in the skeleton execution a specific advice shall be executed.

There are two basic possibilities for managing the scope of weaving instructions: Bundled with skeletons or globally applicable. Weaving instructions can, for example, only be valid for a single composite application. This would mean that a weaving instruction is closely tied to a particular skeleton and not used for other skeletons. This way the base skeleton, a set of weaving instructions and potentially also the used advice constitute a unit that, with all its parts, would implement the complete composite application. When starting the application the skeleton is loaded into the composition execution engine and the respective weaving instruction set would be loaded into the weaving engine for the same composition session. Only this set of weaving instructions would then be used in this composition session and other composite services would use a different set of weaving instructions.

One goal of this thesis is to explore, if AOP mechanisms can also be used for global management of installed applications and in particular of associated functional and non-functional concerns. This would potentially enable the application of policies or new features across all installed applications with minimal effort. Weaving instructions being too closely tied to single applications, do not help in this respect. Therefore, the second alternative for scoping the weaving instructions was chosen: All weaving instructions are global to the composition engine. They are therefore applicable to all deployed skeletons and all composite applications that are executed by that composition engine.

Even if considering weaving instructions to be global to the service environment, it would still be possible to limit the scope of certain weaving instructions to a subset of skeletons. This can be reached by means of the weaving condition within the weaving instruction. It can contain a list of applicable skeleton IDs. Thus, the related advice would only be applied if the triggering join-point is within those skeletons. Furthermore an implementation of advanced weaving instruction management is possible. It may allow a sophisticated and flexible assignment of service scope to sets of weaving instructions. However, this was not investigated in greater detail within the scope of this work. Here weaving instructions are considered to be globally applicable.

4.5.2 Weaving Instruction Syntax and Semantics

This thesis proposes declarative weaving instructions that are always a triplet of 'Condition', 'Control' and 'Advice'. A complete weaving instruction would look as follows:

```
Condition: <ssm based condition statement>
Control: <weaving control hint>, <mode of execution>
Advice: <skeleton id>
```

with

```
<weaving control hint> = BEFORE | AFTER | AROUND
<mode of execution> = SYNC | ASYNC
```

Condition

The condition statement is used in order to evaluate if advice has to be triggered at a given join-point. The condition is expressed based on shared state variables and basic mathematical and logical operators. It is similar to expressions used in skeleton condition elements and uses the same syntax. In fact, in the prototype implemented for this thesis, the same code is used to parse and evaluate condition elements in weaving instructions and constraints in service templates.

In order to express the condition, shared state variables can be used. Thus, the full composition run time session context can contribute to the decision if an advice needs to be applied. On top of the composition session shared state, there are also AOP specific variables introduced that expose further context from the internal run time of skeleton elements which is usually not reflected in the shared state. This is explained im detail in Chapter 4.6. This means that a rich base of contextual information is available allowing highly expressive weaving conditions.

Control

The control statement of the weaving instruction guides the advice execution. It determines the exact details of weaving execution at a join-point and also the mode of advice execution through a weaving control hint. The weaving control hint was already discussed in Chapter 4.4. it specifies the execution of the advice being before, after or instead of the skeleton element that constitutes the respective join point. The proposed syntax of the weaving instruction is to use the keywords 'BEFORE', 'AFTER' or 'AROUND'.

By default the execution of an advice is performed synchronously. This means, that the execution of the target skeleton is halted until advice execution has finished. Asynchronous advice execution would allow that the base skeleton execution resumes, while advice is still executed.

The mode of execution determines if the advice execution is performed synchronously or asynchronously. Asynchronous advice execution might lead to better lead-times in overall composition execution. On the other hand it bears the danger of creating race conditions if the advice or any action initiated from the advice will interfere with the main composition session. For example, both would write into the same shared state variable. A good rule would be to only allow independent actions executed in the advice and make all data from the base skeleton read only for the advice execution session. The mode of execution is specified using the keywords 'SYNC' or 'ASYNC' after the weaving control hint separated with a comma. The mode of execution is optional and if missing it would default to synchronous execution.

Advice

Advice is implemented by means of another skeleton and the composition execution engine also executes the advice skeleton. These skeletons are not special in the sense that they follow the same syntax as any other composition skeleton. They are deployed, managed

Join-Point	Keyword used in the variable aop.joinpoint
Start Element Join-Point	start
End Element Join-Point	end
Service Template Join-Point	template
Service Selection Join-Point	selection
Service Invocation Join-Point	invocation
SSM Command Join-Point	ssm
Condition Element Join-Point	condition
Goto Element Join-Point	goto

Table 4.1: Keywords for identifying join-points in weaving conditions

and identified in the service environment as like any other skeleton. Consequently, advice in the weaving instruction is always specified as reference to a skeleton using its unique skeleton ID.

4.5.3 Specifying the Join-Point

An essential part of a weaving instruction is the ability to specify the targeted join-point. Here it is proposed to not have an extra fourth element within the weaving instruction, but to rather include this in the condition statement. For this purpose the composition engine exposes a special variable to the weaving engine that indicates the currently processed join-point:

```
aop.joinpoint = <the currently processed join-point>
```

In the condition statement of the weaving instruction this join-point variable can be used to filter out the right join-points at which advice shall be applied. An extra condition based on this variable and the keywords from Table 4.1 are connected with logical conjunction to the other conditions.

This method means that at every join-pint always all weaving instructions are considered. The advantage is that the weaving instruction's condition expression is the only contributer to the decision if the advice shall be applied. The disadvantage is a potentially high effort of always checking all weaving instructions at every potential join-point. This is further explored and discussed in the validation in Chapter 5.

4.5.4 Weaving Instruction Skeletons and Weaving Modules

How the weaving instructions are uploaded into the composition engine and how they are managed is a small but important implementation detail. Using XML based specification and uploading it into the engine through an extended management API would be a feasible solution. However, this thesis proposes a different method. It rather proposes to upload weaving instructions by means of a skeleton that is containing a new skeleton element

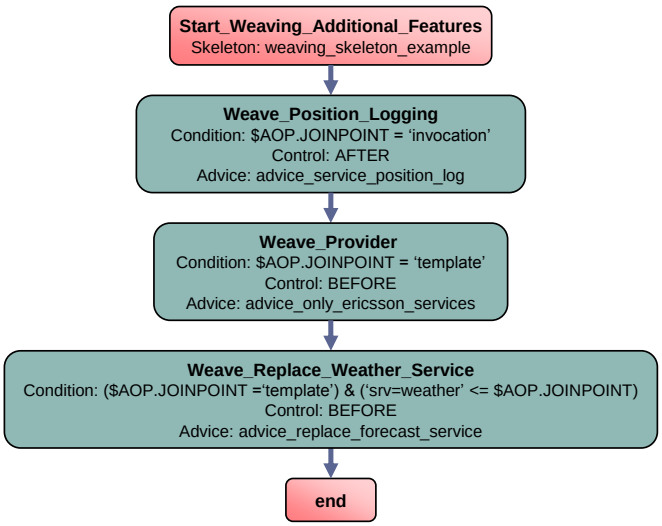

Figure 4.6: A skeleton that contains weaving instructions.

called 'weaving instruction element'. The weaving instruction element always consist of three skeleton element parameters reflecting the weaving instruction triplet.

Figure 4.6 shows an example skeleton that contains weaving instructions. It contains three weaving instructions for three different purposes. Please note the use of the condition statement in this example. In all three weaving instructions, a condition is used, which specifies the join-point. The first two weaving instructions match each join-point of the respective type.

The third weaving instruction's condition contains an additional criterion. A match also depends on the constraint specified in a service template that corresponds to the join-point. The overall condition is true, if the currently processed join-point is a service template, where the constraints expression contains the string 'srv=weather'.

When executing a weaving skeleton or more specifically when executing any weaving instruction skeleton element, the composition execution engine uploads the weaving instruction details into the engine's configuration. It would then immediately be considered by the weaving engine for executed composition sessions.

The concept of defining weaving conditions from skeletons provides a couple of interesting features. First of all this allows modularization if only weaving instructions that belong together are put into one skeleton. For example, all weaving that belongs to one specific policy can be grouped within one skeleton. Thus, several distinct policies would lead to a number of specialized weaving skeletons. Therefore, weaving skeletons constitute weaving modules.

Weaving skeletons can contain all other skeleton element types. Using, for example, the goto element would allow calling other weaving skeletons in order re-use them. Service templates could be used for querying additional services that may help decide if a particular weaving instruction should be loaded or not. In this respect also the condition element is useful for creating alternative branches in the weaving instruction specification.

The skeleton based aspect definition and upload opens up many possibilities for aspect management as discussed in Chapter 3.

Further management of the aspects would be done within an extension of the service creation environment that combines already a skeleton editor, service description design, deployment and debugging facilities. It was extended to allow selective disabling and removal of aspects from the engine. Furthermore it allows detailed tracing and debugging of advice execution sessions within composition sessions.

4.6 Data Exposure Towards Weaving and Advice

Information from the composite application execution is presented to aspect weaving and execution. In principle all information that reflects the application's general business logic and data of the execution run time is potentially a target for advice induced modifications and the base for weaving decisions.

The following data reflects the view of aspects on the composite application and its execution:

Shared State:
The shared state contains run time data from the composite application's business logic. It is directly available to advice logic and weaving.

Skeleton Element Parameters:
The parameters of a skeleton element are accessible to aspect logic through predefined shared state data structures.

Execution Engine internal data:
Internal run time date of the composition execution engine is usually not reflected in the shared state. As it contains key data with respect to identification of join-points and constituent service execution, it is made available to aspect logic through newly introduced additional shared state variables.

In AOP execution there are two main uses for data about the composite application: First of all data is available for evaluation of weaving instruction. Related variables can be used directly in formulating weaving conditions. Furthermore, data is available in advice execution. This means it can be used in the logic of advice skeletons and it can be modified by the advice. In the proposed implementation, the same data is presented in both cases, thus, the data used in weaving evaluation is also available to the advice when it is executed. This chapter explains the details of which data can be accessed and how it is exposed by the composition engine towards the AOP sub-system.

4.6.1 Exposing the Shared State to Weaving and Advice Execution

Advice is started with its own shared state session. It contains a full copy of the data from the base session's shared state. Therefore, all data of the composite application is available to the advice. If the advice however writes into its shared state, the modifications stay encapsulated within its own session. This behavior is important in order to avoid unwanted interference from advice execution into the composite application and other advice that is executed before or after. Each new advice session gets a copy of the most recent base session's shared state at the time of reaching the join-point.

Regardless the encapsulation of the advice execution within an own shared state session, advice needs to have the possibility to apply selective modifications to the base composition session. For this purpose the base session shared-state can be accessed by means of explicitly denoting the parent session at variable access.

For example, if the SSM command statement

```
ssm_var = 'example'
```

is used from within an advice execution session a value is assigned to a variable. This write stays local within the advice session. If instead the same advice uses

```
parent.ssm_var = 'example'
```

the value is written to a variable with the same name within the parent composition session. Therefore, the entire parent session's shared state can be modified from the advice. The need of using the parent method for doing so means that the developer can easily control which writes need extra considerations regarding aspect inter-work while local variables can easily be used independent of other aspects or base application logic.

If there are several consecutive advices executed at a join-point, the changes applied to the base shared state by one advice will persist and be presented to the following advice execution session within its initial shared state copy. As the weaving condition of advice is evaluated based on the shared state, changes applied by one advice can therefore influence the invocation of subsequent advice. This can be intended, but it also bears the danger, that one aspect unintentionally breaks another aspect. Therefore, managing and controlling advice interactions is one of the most critical tasks in an AOP enabled environment.

4.6.2 Exposing Composition run time and Skeleton Data

As the join-point model is based on skeleton elements and weaving is based on monitoring of skeleton element execution, the availability of composition execution run time data is essential for aspects. Shared state contains run time data available to composition logic. It is exposed to the AOP sub-system by means of dedicated shared state sessions as explained in Chapter 4.6.1. What is missing is a mechanism that enables advices and weaving to

make use of internal data of the composition execution engine. Not only reading this data, but also modifications should be possible in order to allow fine-grained influence on composition execution.

An example of significant run time data from within the composition execution engine is the list of service candidates returned by the service database at service selection. Another example would be the expressions being used as skeleton element parameters. Examples are the service selection constraints or the service invocation parameters. The variable aop.joinpoint mentioned in Chapter 4.5 is yet another example. It specifies the current join-point and is therefore highly important for weaving instructions.

This thesis proposes the introduction of a reserved name-space in the shared state for AOP related purposes. The main purpose is keeping variables that reflect the composition execution engine internal run time data. This means that the composition engine internal data is exposed to the AOP environment also by means of shared state. The weaving engine and advice can access and utilize this data through SSM commands and SSM variables as like any other data in the shared state.

The reservation of a reserved name-space is done by introducing the reserved keyword 'aop.' that will precede all AOP specific variables. All aop related variables are therefore addressed as:

```
aop.<variable name>
```

The variable aop.joinpoint is the first example. When a new shared state session is created for weaving evaluation and advice execution, a number of these AOP specific variables is created based on composition engine run time and skeleton element parameters. Thus, shared state for AOP will contain a copy of the parent session's shared state plus all these AOP specific extra variables. This mechanism is relatively flexible. Future extensions exposing more internal run time data of the composition execution environment can easily be introduced with this method.

Which AOP specific variables are created and added to the shared state depends on the join-point. Always available is aop.joinpoint. The variable aop.constraints is, for example, only added at join-points that are related to the service template or the start element of a skeleton. Only these skeleton elements contain a constraint parameter, thus, only for them it can be provided. In this respect the list of candidate services is only available in the service template related join-points Service Template, Service Selection and Service Invocation.

Some of the AOP specific variables can only be read while others also allow an advice to perform writing operations. In general, writing into AOP specific variables is always formally allowed. It will not lead to error, but it will not always have an effect on the composition run time. If a variable can be modified by advice, depends on the join-point and to a great extend also on the related weaving control. The service selection constraint is a good example. It is only available for modification until it is used for performing service selection. After this is done, the constraints cannot be changed any more through the respective AOP variable. The reason is not only that changes to the constraint cannot

Join-Point	Weaving Point	Read/Write access available
Start Element Join-Point	After	read only
End Element Join-Point	*	n/a
Service Template Join-Point	Before After Around	read/write read only read only
Service Selection Join-Point	Before After Around	read/write read only read only
Service Invocation Join-Point	Before After Around	read only read only read only
SSM Command Join-Point	*	n/a
Condition Element Join-Point	*	n/a
Goto Element Join-Point	*	n/a

Table 4.2: Read/Write availability of the variable aop.contraints

effect the service selection any more, but also to keep the constraint value that has actually resulted in the present list of service candidates. Thus, the cause and the result of that action are kept consistently reflected in the run time data.

Advice being executed before the service template can overwrite the constraint. The modification will be taken into account for the service selection process that follows after advice execution. The new constraints value will also be considered as input to subsequent weaving and advice execution at the same join-point. It can therefore be modified multiple times before the service template uses it. Advice after the service template or just after the service selection will still be able to read the constraints, but any modification would be ignored. Table 4.2 shows the complete read and write possibilities for this variable. Not applicable means that the variable is not created when the shared state is prepared.

AOP specific shared state variables are only available in shared state created for advice execution and weaving. The base shared state of the composite application will never contain them. Furthermore, AOP specific variables expose parameters of the skeleton element and advice can potentially modify them. These modifications are not persistent. They are only in effect within the scope of execution of the skeleton element for which they are applied. After execution proceeds and leaves this skeleton element, all modifications to the skeleton element data is discarded. The next time this skeleton element is executed, all settings are back to what the default as specified by the base skeleton. If modifications are wanted again, the respective advice needs to be executed again.

If asynchronous advice execution is selected, all data is exposed to the advice read-only. Modifications are too dangerous with respect to race conditions. Thus, all remarks regarding writing into AOP specific variables are valid only for synchronous advice execution sessions. The following sections introduce syntax and semantics of important AOP specific

variables.

4.6.3 Join-Point Type

```
aop.joinpoint = <join-point specifier>
              = start|end|template|selection|invocation|condition|ssm|goto
```

At all join-points, the type of the join point itself can be found in this new variable. For example, at the service selection join-point this variable contains the value 'selection'. Table 4.1 shows the possible values. This information is important for the weaving condition, as it allows specifying at which join-points the advice is applicable. Thus, it allows to introduce a join-point filter in weaving. This variable is only reporting context information, and thus, exposed read-only.

4.6.4 Branching Condition

For the condition skeleton element the condition expression is the essential data to be exposed as special AOP variable:

```
aop.condition = <condition expression string>
```

This variable is available for condition element join-points. If weaved in before this join-point, the advice can modify the condition, and thus, change the behavior of the composite application. After the condition element the result of the evaluation is available using the following variable:

```
aop.condition_result = <result string>
```

It cannot be changed any more, but it allows applying advice as first operation after the condition element in some of the skeleton branches. The branches are named in the skeleton and this variable exposes the name of the chosen branch. If, for example, the condition results in a boolean result, the name strings 'TRUE' and 'FALSE' are used.

4.6.5 Constraints

```
aop.constraints = <constraint expression string>
```

This variable served as example above with a detailed description. It contains the constraints of a service selection or start elements. The variable is therefore available at the service template, service selection, service invocation and start join-points. Only advice that is executed before service template and service selection join-points can modify it. The modified value would then be considered in the following execution of the skeleton element.

4.6.6 Results of Service Selection

The following variables contain the results of the service selection process within service template execution:

```
aop.service_candidates = <list of service IDs string>
aop.candidate[<n>] = <service ID sting>
aop.service_candidates_count = <number of service candidates>
```

These variables allow access to the list of service candidates as returned by the constraint based query to the service database. They are therefore available for advice that is executed after service selection was performed. More specifically this is after the service selection or the service template join-points and before and after the service invocation join-point. The variables are read only with the exemption of after the service selection join-point. Here the candidate list can be modified by advice in order to enforce a different selection result.

The variable aop.candidate is an indexed list allowing to access each entry individually. It contains the ID strings of service candidate as returned from the service database. Therefore, modifying the list means that another service ID strings would need to be written. The number of service candidates is provided by the variable aop.service_candidates_count. For example:

```
aop.candidate[0] = 'service_01'
aop.candidate[1] = 'service_02'
aop.candidate[2] = 'service_02'
aop.service_candidates_count = 3
```

The variable aop.service_candidates contains a serialization of the candidates list:

```
aop.service_candidates[0] = 'service_01,service_02,service_03'
```

Writing into this variable allows to write a service candidates list in one operation.

4.6.7 Parameters of a Constituent Service

The invocation parameters for constituent services are specified in the service template. These parameters are exposed by means of the special variable aop.param.

```
aop.param[<n>].key           = <parameter id>
aop.param[<n>].expression    = <parameter expression string>
aop.param[<n>].value         = <parameter value>
aop.service_parameters_count = <number of service parameters>
```

The variable aop.param is an indexed list. Each list element corresponds to one service invocation parameter. Each parameter in aop.param is presented by means of a key, the parameter expression and the parameter value. The key contains the identifier or name of the parameter. Expression is the string that defines the value used at service invocation. The value field of the aop.param variable contains the actual value that is the result of evaluating the expression. Advice can write into aop.param at all weaving points in the context of the service template before the service is actually invoked.

Writing into the key changes the identifier of the parameter. The value that is used at service invocation can be influenced from the advice by changing the expression. The value part is read only. By writing aop.param the entire API of the constituent service can be changed. Together with a change of the service selection constraint the entire service template can be re-programmed by an advice to select and invoke an entirely different constituent service.

The variable aop.serviceparams is another way to expose the service parameters:

```
aop.serviceparams.<parameter id> = <parameter expression string>
```

Instead of a numbered list of parameters it includes the key into the variable hierarchy. This method allows a more intuitive access to the parameters. On the other hand it only allows to apply modifications to the parameter expression. The number and identifiers of parameters cannot be changed. The following example shows both methods exposing the same parameter:

```
aop.param[0].key        = 'user'
aop.param[0].expression = '$(USERID)'
aop.param[0].value      = 'joerg'

aop.serviceparams.user  = '$(USERID)'
```

4.6.8 Invoked Service

After service selection one service candidate is selected for invocation. If no advice interferes, this would be the first service from the list of service candidates. The variable aop.selected_service provides this information to the AOP environment.

```
aop.selected_service = <service ID string>
```

This variable is available at the service invocation join-point and after the service template. At the weaving point before service invocation it can be modified in order to enforce a different service to be invoked.

4.6.9 SSM Operations

Each SSM command can also be subject to weaving and modifications. It allows to override changes to shared state variables. This can be reached by modifying the expression of the SSM command before it is executed. The following variable exposes the expression string:

```
aop.ssm_comman_expression = <ssm command expression string>
```

Write operations into this variable are available to advice being executed before the SSM Command. There is however always the possibility that advice writes directly into the same shared state variable that is also written by the SSM command.

4.6.10 Skeleton Element Identification

Advice might be wanted only at particular skeleton elements. A variable is introduced, that exposes the identifier of the skeleton element:

```
aop.skeleton_element_id = <ID of the skeleton element>
```

This variable is available at all join-points. It has purely an informative character and is therefore read only. It can be used to filter out particular skeleton elements in weaving.

4.7 Advice Selection and Execution

This chapter explains the details of how advice is managed and executed by the AOP enhanced composition engine. For this purpose the architecture of the Ericsson Composition Engine is extended by introducing a weaving engine as shown in Figure 4.7.

Online weaving is based on some kind of run time monitoring of the composition execution in order to detect when to apply advice. This monitoring needs to be made aware of each join-point and weaving-point that is reached. Join-points and weaving point online detection is naturally allocated in the composition manager where a skeleton is executed. The evaluation of weaving instruction and invocation of advice is handled by the newly introduced weaving engine. The weaving engine contains and manages a global repository of weaving instructions.

When the composition manager reaches a weaving-point in the skeleton execution, it throws a weaving point specific event. It does this unconditionally for each weaving point that is reached. The event manager was also implemented together with the AOP solution, but it is used also for other purposes than invocation of join-point handling.

From the point of view of the event management the weaving engine takes the role of an event handler that is subscribed to the AOP specific join-point events. The weaving engine also contains all weaving instructions or the global point-cut. Each of the join-point events triggers the evaluation of weaving instructions. In case of a match the composition manager is ordered to execute the respective advice.

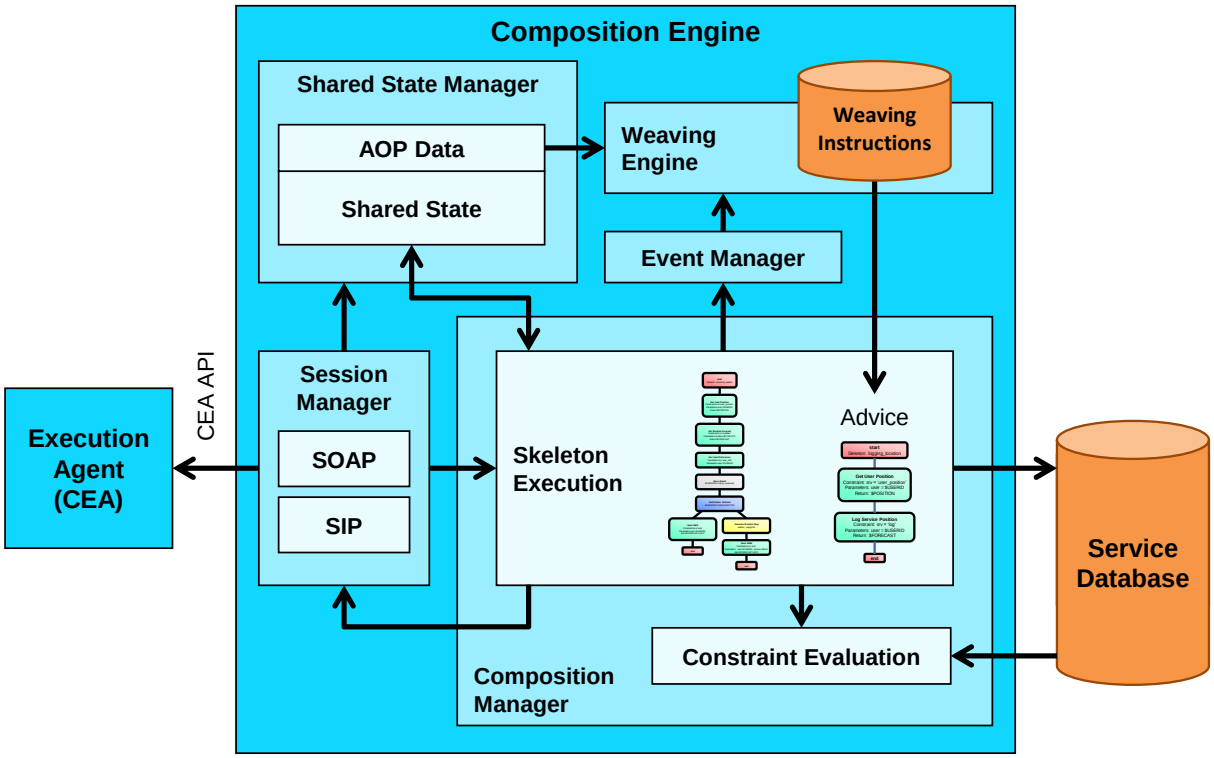

Figure 4.7: The internal structure of the Ericsson Composition Engine extended for aspect weaving based on event management

Event handling is synchronous, thus, the execution in the composition manager waits until the weaving manager has finished processing the weaving-point. This includes the weaving checks and also the advice execution. This means that the composition manager executes an advice session while the related base skeleton session is on halt.

Events can bear data. This is used in order to communicate information of the local weaving-point execution context, for example the type of the weaving point and the shared state session id. This data is used to build the special AOP shared state sessions, through which all data is presented to the weaving evaluation logic. This same AOP shared state session is also directly used by the composition manager for advice execution. From the point of view of the composition manager the advice execution is not different from any other skeleton execution. The connection to base session data is maintained through the special shared state variables in which the advice execution in performed. The shared state manager was extended to handle the potential write-trough to the data of halted base session.

The AOP shared state enables advice operations that cannot be performed from an ordinary skeleton. For example, SSM commands can access and change constraints or parameters for service invocation. In this respect, it depends on the join-point context, if a certain type of parameters is available to the advice logic.

The AOP solution is implemented entirely in the service technology agnostic part of the

composition engine. The join-point model is based on technology agnostic skeletons and all AOP applied decisions are taken before a constituent service is invoked. The technological details of service execution are intentionally out of reach of the AOP environment because they are also out of reach for the composition manager. This is an important conceptual consistency: the AOP environment inherits the technology-agnostic behavior of the composition engine. Operation within a heterogeneous service environment is therefore fully supported also when using AOP.

4.7.1 Skeleton Based Advice Implementation and Execution

Advice is in AOP implementations often considered to be code fragments, which are weaved into the target code creating an updated source code. For the Ericsson Composition Engine, advice is implemented by means of an additional skeleton that is executed within a dedicated composition session. All operations of a skeleton are therefore available for advice implementation. This includes among others: conditional branching, modifications to shared state variables and execution of dynamically selected services. Advice skeletons are also deployed in the service database as like any other skeleton. Advice skeleton execution is similar to the execution of an explicitly started sub-skeleton, but there are also important differences: advice skeletons have access to additional AOP related variables. Consequently, advice skeletons using AOP related shared state variables can, in general, not be used as ordinary composite services. Trying it would not necessarily lead to errors, because the shared state manager instantaneously generates new variables. Nevertheless, these variables might not contain valid data, because they do not originate from a join-point context.

Using skeletons for implementing advice offers many options to apply changes to the base skeleton. It is, for example, possible to reason about the change that is needed within the advice and also use external services for doing so.

There are several ways to apply a behavior change to the target composite application. It is, for example, not necessary to replace an entire service template if another constituent service is wanted. A modification of an existing service selection constraint might be enough. If the new service has a different API, also invocation parameters and the variables that receive the service execution result can be changed. Technically the same service template is used, but is re-programmed by the advice.

The the proposed skeleton based advice execution based on separate shared state sessions means that advice becomes a separate process. From service composition perspective it contributes additional reflective constituent services composed into the overall service serialization of the composite application. The special reflective role of advice in contrast to other constituent services is the possibility to not only contribute additional functionality, but to divert from the application's original run time behavior as defined by its base skeleton. Advice is therefore a special constituent service that can re-program the composite application is became a part of. In this respect the weaving engine constitutes a second composition method within the Ericsson Composition Engine.

4.7.2 Advice Execution Order

In the proposed join-point model each join-point may correspond to several weaving points. At each of the weaving points individual weaving decisions are taken and advice is executed independently from the other weaving points. The following rules determine the weaving order:

- Each join-point and weaving point is handled separately in the order determined by base skeleton execution. For example, advice weaving for an AFTER weaving-point at one a join-point is executed before the BEFORE weaving point of the following join-point.
- Within a join-point the weaving points are considered separately. First the weaving and execution of all advice is done for the BEFORE weaving point. Then, weaving and advice execution for AROUND is is performed. Finally, after all advice for AROUND weaving is finished, the weaving and advice execution for the AFTER weaving point is done. An around weaving might inhibit the execution of the respective base skeleton element for this join-point, but the after weaving is still being executed.
- If several advices shall be weaved around a join-point, they are all executed in the order of their weaving instructions. Thus, they all together they replace the original skeleton element.

These rules for advice execution sequence are demonstrated in Figure 4.8.

Within the scope of this work no specific control of advice execution order at a weaving point was introduced. This means if several advices need to be executed at the same join-point, it is done in the same order they were found in the weaving instruction checks. This ultimately means that the order in which weaving instructions were loaded determines advice execution priority. More elaborate control mechanisms can be implemented and used for managing advice-interaction. For example, the priority field in the skeleton start element of weaving skeletons can be used. Advice may then be invoked one after the other in the order determined by that priority information. Furthermore the priority setting could be done in the aspect administration front-end.

4.7.3 Structural Modifications

With the chosen join-point model and advice execution paradigm, the possibility to apply structural changes to the skeleton execution flow is limited. Advice can be seen as additions to the execution flow, but it is encapsulated within an own session, and therefore limited to local changes in shared state and run time behavior. This does not change the global structure of the composite application's base skeleton.

It is possible to influence how single existing base skeleton elements behave. For example, conditional execution can be changed by modifying the guiding condition, but it is

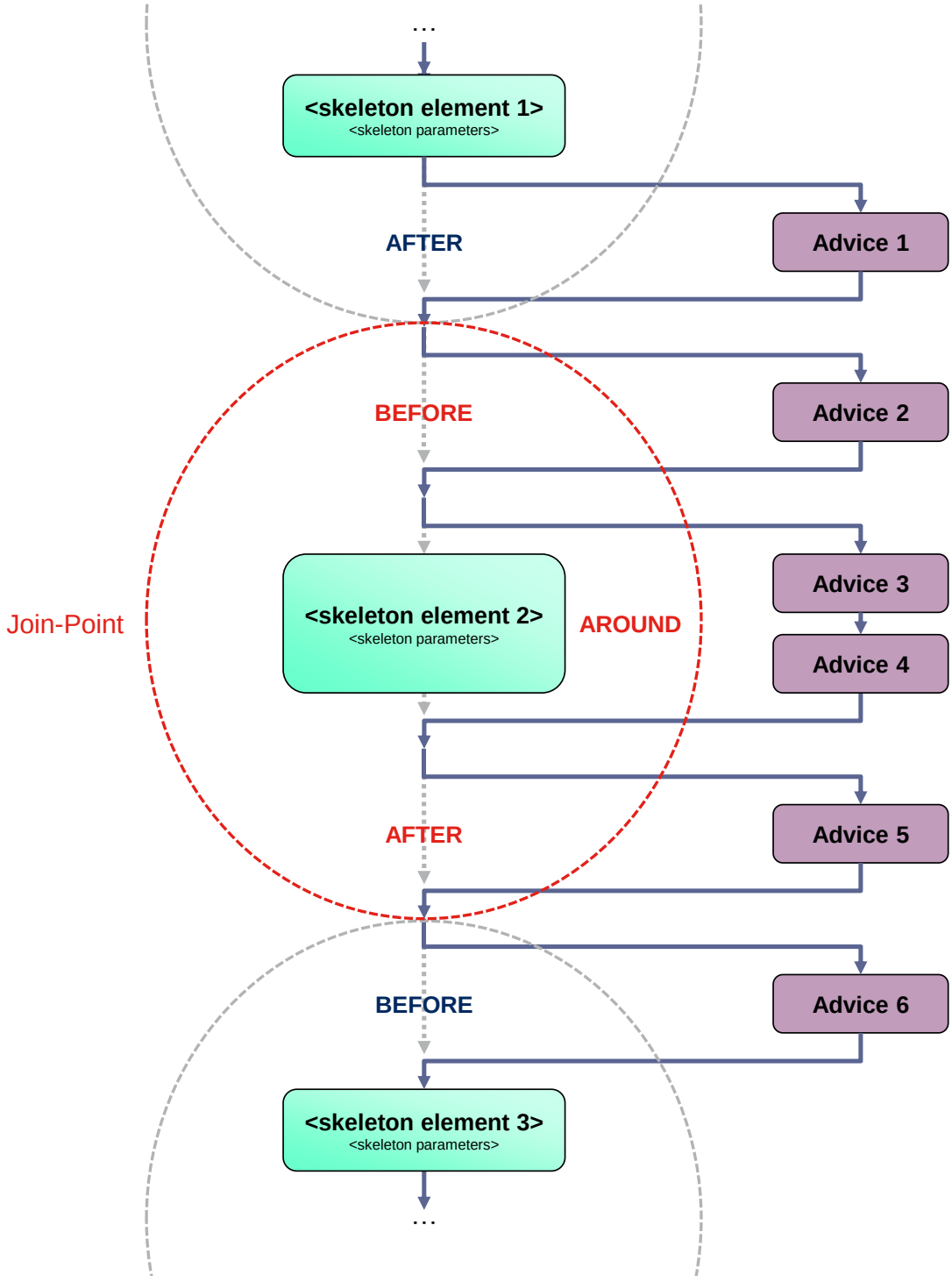

Figure 4.8: Advice execution order between and around skeleton elements

not possible to create or remove new skeleton branches or add new skeleton elements. By means of AROUND weaving control it is possible to remove or replace service templates, but this is a local modification that does not alter the global structure of the skeleton.

Structural modifications applied through advice would require first of all a different join-point model. It needs to capture the semantic elements of the skeleton language that define it's structure. In the skeleton language of the Ericsson Composition Engine those would mainly be connections between the skeleton elements. Even though they are not named, identifying these connections could be done by specifying the combination of originating and target skeleton elements. This would constitute a new join-point type. If these connectors would be exposed to the weaving evaluation, they could be part of the weaving condition. Furthermore, advice could, for example, determine explicitly in with which skeleton element the execution shall be continued.

While it would be possible to create the respective semantics and syntax needed to apply elaborate structural changes through aspects, the question remains if they would be needed. Knowing that the base skeleton's general structure would not change, still provides a good common foundation for developing multiple aspects that address different concerns, with well controlled interaction between the aspects. If the common structural base becomes variable, Aspects can more easily break each other's functionality. They become more dependent on each other's implementation details. Consequently, the separation of concerns in the development process becomes worse.

It is also questionable if the availability of extensive structural change would have any major practical value. Additional services can already be added to a composition or removed from it with the proposed AOP system. This targets the main purpose of service composition flexibly: to orchestrate and coordinate the execution of a set of constituent services. Even new conditional branches in the execution flow can be realized by either branching within an advice or by using a combination of several related advices.

Other implementations of AOP provide similar features for applying modifications. Aspects can apply changes to a mostly constant structure of the base application rather than allowing extensive re-programming. This makes sense, as one of the main targets of AOP is the reduction of application complexity. Complex interdependencies between aspects introduce new code entanglement rather than reducing it.

This brief discussion suggests that the vast majority of practical use cases can already be covered with the proposed join-point model, weaving logic and advice capabilities. More complex aspect induced modifications will, in general, not lead to better implementations.

4.7.4 Inter-Advice Communication

Aspects often do not consist of only a single advice. Multiple advices may need to work together in order to implement a concern throughout the application. This can be multiple instances of the same advice at multiple join-points, or it can be different advices that together implement the concern. In general, these cases require communication between individual advice instances.

One possibility to reach inter-advice communication would be by using an external

service containing a database. It would be used to store common data of several advice instances. Then all advices would implement read and write methods towards their common data storage in the external database.

Shared state variables is persistent within the execution session of the composite application. Thus, it also persists across advice execution sessions. This means one advice might create and write into a new shared state variable and another advice is then able to read from that same variable. This way advice instances share their state with each other, and thus, communicate. A sub-set of the shared state becomes run time data of the multi-part aspect implementation.

A shared state based solution works for advice communication within a single instance of the composite application. If aspects need to exchange data between execution instances of the composite application, the method concerning an external common database can be used.

4.7.5 Weaving Loops and Their Prevention

Weaving is global within the composition engine and can, in general, be applied to all skeletons. As advices are also skeletons, this means that they are themselves subject to weaving. This behavior is conceptually wanted, but bears the danger of creating loops. For example, a weaving instruction might catch all service selection join-points and the related advice might contain a service templates. In this case, new instances of the same advice will continuously be weaved into itself. This kind of loop must be avoided.

The following loop prevention is proposed: With each weaving a new shared state session is created for advice execution. The session, from where the weaving was initiated is the parent of that advice execution session. Multiple nested weaving will lead to a stack of nested parent-child relationships and respective shared state sessions. A loop prevention rule forbids weaving of an advice, if a shared state session for that same advice is already in the current session stack. This is the solution implemented in the prototype. A simple alternatives may be to forbid nested weaving in advice execution sessions altogether.

Chapter 5

Validation

The Aspect Oriented Programming environment developed by this dissertation provides an additional tool for implementation of cross-cutting concerns to software developers. Nevertheless, all features implemented by means of Aspect Oriented Programming could also be implemented without it. A solid recommendation when and how to use AOP requires a good understanding of related advantages and disadvantages compared to the targeted use case domain. The validation of this thesis studies the detailed properties of a reference implementation of the proposed concepts with respect to the hypothesis presented in Chapter 1.7. In order to validate the hypothesis, two central questions need to be answered:

1. **Can typical use cases be implemented using aspects?**
   The first evaluation criterion is, if the proposed AOP system is suitable to implement typical use cases and if cross-cutting is considerably reduced when using the prosed AOP solution.

2. **Is the performance acceptable?**
   The second evaluation criterion is about latency in service response times that comes from the presence and usage of an AOP solution.

The starting point of the evaluation is a demonstration of typical use cases in Chapter 5.1. The idea is to demonstrate the practical use of AOP in order to solve various typical use cases. Naturally the selected use cases are all related to modifications or extensions of composite applications. This will provide a good overview of the possibilities a developer has and how practical AOP works for solving typical implementation tasks.

The special attention of this thesis on telecommunication service applications implies severe requirements on performance. Especially any changes to end-to-end response times or impacts on session set-up and signaling latency are critical. In order to investigate this, the live performance of the AOP environment is investigated in Chapter 5.2. It provides a comparison of application performance if a concern is implemented with and without using AOP. It furthermore explores the limits of the AOP environment with respect to performance. This tries to answering the question of the extent to which AOP can be

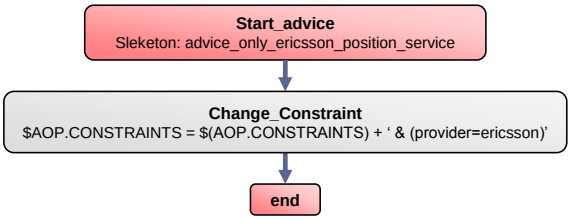

Figure 5.1: Advice for modifying the service selection constraint.

used until the execution engine's performance is too severely impacted. Measurements are performed using the prototype implementation of AOP sub-system within the Ericsson Composition Engine.

From the performance measurements a theoretical execution model of the AOP enabled composition engine is build. This model is used in Chapter 5.3 for exploring the characteristics of alternative AOP feature sets and their implementations. There is a natural trade-off between performance and richness in AOP features and this thesis provides quantitative insights for finding the optimal AOP solution for an application domain.

5.1 Typical use cases implemented with AOP

Will the proposed AOP techniques be useful for implementing common practical use cases? This chapter investigates this question by demonstrating how implementation challenges of composite applications are solved without using AOP compared to using the suggested AOP framework. The examples of Chapter 2.2.12 are used as reference. They represent a non-AOP implementation. This chapter describes how typical modifications are implemented using the proposed AOP framework.

All examples show simplified applications in order to demonstrate certain techniques. A product grade composite service would typically address many more concerns.

5.1.1 Example: Preferred Service Provider Policy

The main task of service composition is selection and controlled execution of constituent services. Therefore, applying modifications to the service selection will be the single most frequently used target for aspects. For example, the owner of the composite service environment might have a policy to only allow constituent services coming from a particular service providers. This limits the selection of constituent services and can be introduced using additional selection constraints. It also constitutes a globally cross-cutting concern, because selection and execution of the respective constituent service might be implemented throughout all deployed composite applications.

The implementation of this concern can be done by means of advice that is modifying the service selection constraint. The advice shown in Figure 5.1 adds the constraint 'provider=ericsson' to the already existing constraint expression of a service template. It

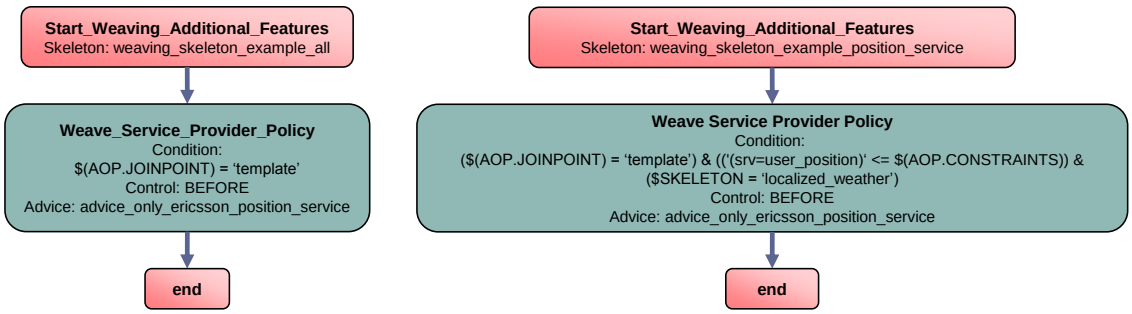

Figure 5.2: Weaving instructions for applying advice to all service templates, or to those where a particular constraint is used

is doing so by utilizing the special AOP variable that exposes the constraints expression in the shared state. When the service selection is executed, the new constraints expression would be considered, leading to a list of service candidates that meet the additional provider constraint.

The advice needs to be applied by a weaving instruction that determines if and where the modification is applied. Figure 5.2 shows two possible alternative weaving instruction skeletons for applying this advice. The left skeleton defines a weaving instruction that catches all join-points of type service template. As this is the only condition, the weaving engine following this instruction would apply the advice globally to all service templates in all composite applications. Thus, the service provider policy is applied comprehensively. Only a single weaving instruction and a simple advice skeleton is needed to do so. This demonstrates how efficient and expressive AOP can be with respect to distributing globally cross-cutting modifications.

The weaving instruction on the right side of Figure 5.2 filters out all service template join-points where the sub-string '(srv=user_position)' is used in the constraints expression of the respective service template. The constraints expression string is available by means of the special AOP variable "AOP.CONSTRAINTS". Furthermore, the weaving condition catches if the skeleton 'localized_weather' is currently executed. This information is by default available in the shared state of the composition session. It applies the same advice as the other weaving expression, but only to the selection of user position services within the localized weather service shown in Figure 2.18.

This example shows that weaving can have a global scope, but it can also be used highly selectively. Weaving conditions are based on all the extensive information that is available through the shard state. This verifies a high flexibility with respect to weaving scope. It also demonstrates that modifications to the selected constituent services can be applied with minimal effort.

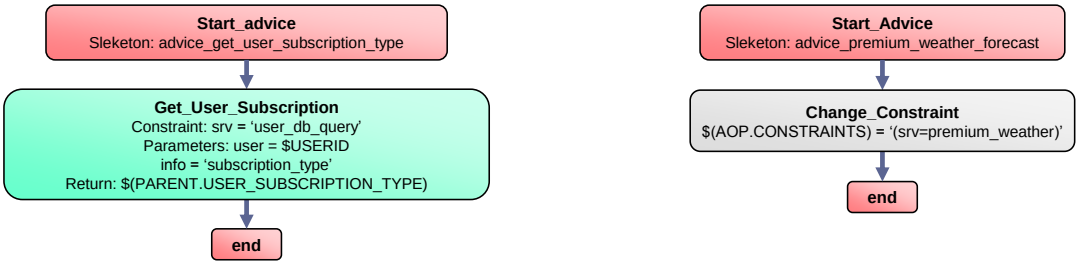

Figure 5.3: Advices for retrieving the user subscription type and for modifying the selection constraints for the weather forecast service

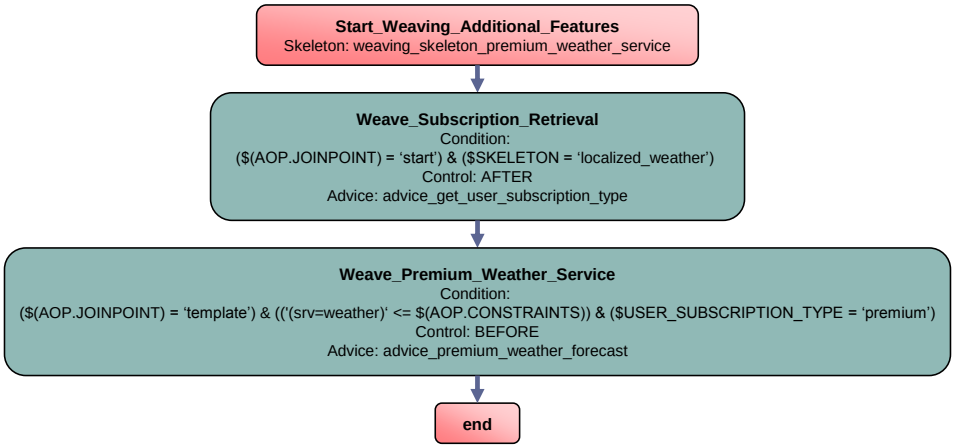

Figure 5.4: Weaving Instructions for retrieving the user subscription type and for changing the weather service used for premium subscribers

5.1.2 Example: Subscription Dependent Service Selection

The previous example shows basic weaving instructions based on the internal shared state. Additionally, modifications can be based on external information sources. An example would be that the service being selected shall depend on the user's subscription. For example, for users with a premium subscription a premium constituent service shall be used. The localized weather service from Chapter 2.2.12 shall be modified in order to provide an extended weather forecast to premium subscribers only. This requires to use a different weather service by selectively changing the service selection constraint.

The information if a user is a premium subscriber is usually not available in the shared state. It has to be requested, for example, from a user profile database. Consequently, weaving instructions cannot utilize subscription information directly for selective advice invocation. The solution would be another advice that first creates this additional information in the shared state.

Figure 5.3 shows the two advices being used and Figure 5.4 shows the weaving instructions that apply these advices. The left advice skeleton retrieves the user subscription type and writes the result into the shared state of the composition session. It is weaved in

by the first weaving instruction after the start join-point of all localized weather services. This ensures, that the shared state of all localized weather services always contains the subscription type.

The right advice in Figure 5.3 replaces the constraint expression in order to request a premium weather service. The second weaving instruction applies this advice before a service template join-point if the original constraint expression requires a weather service and if the subscription type indicates a premium subscription.

Both advices together implement the wanted feature. The data created by the first advice creates information that is used in weaving of the second advice. Thus, the second advice is only executed if the change is actually needed.

This example demonstrates how multiple advice instances can exchange information through shared state and work together in order to reach a common goal. Common access to a global shared state is essential for this purpose. It also demonstrates the addition of a constituent service to the composition from within an advice. Both are fundamental tasks for modifying composite application implementations. This example verifies easy implementation of these modifications with the proposed AOP framework.

5.1.3 Example: Charging a Different User

A common and important concern of service providers is to charge their customers for service usage. This example will demonstrate how to add a dynamic charging and billing solution to composite service applications.

The basic assumption for charging of composite services is that all constituent services are charged for individually [87]. Thus, the sum of all these separate constituent service charges may define the price of the entire composite application. The customer, who has bought the composite application would then be charged for the usage of a set of services rather than the single application that was used from their point of view.

Service providers usually market the applications they offer as consistent units. Implementation details, such as the used constituent services, should usually not be visible to the customer. Consequently the customer shall be billed only for the composite application rather than for the constituent services. As long as all constituent services are anyway free of charge or if they are provided by the same service provider who also owns the composite application, this can easily be solved by the operator's own charging infrastructure. But there might be 3rd party constituent services involved.

This example implements an aspect that hides the actual user of the composite application from the 3rd party service provider. Instead user credential that identify the service provider as user are applied. This way the charges for using the constituent service are directed to the service provider account rather than the end-user, who has bought the composite application.

The knowledge, for which applications this user proxy shall be applied is not hard coded in the aspect. It can be configured externally within a service provider catalog. This information is queried by the advice. More specifically the advice first retrieves all data about a selected service from the service database. This includes the id of the

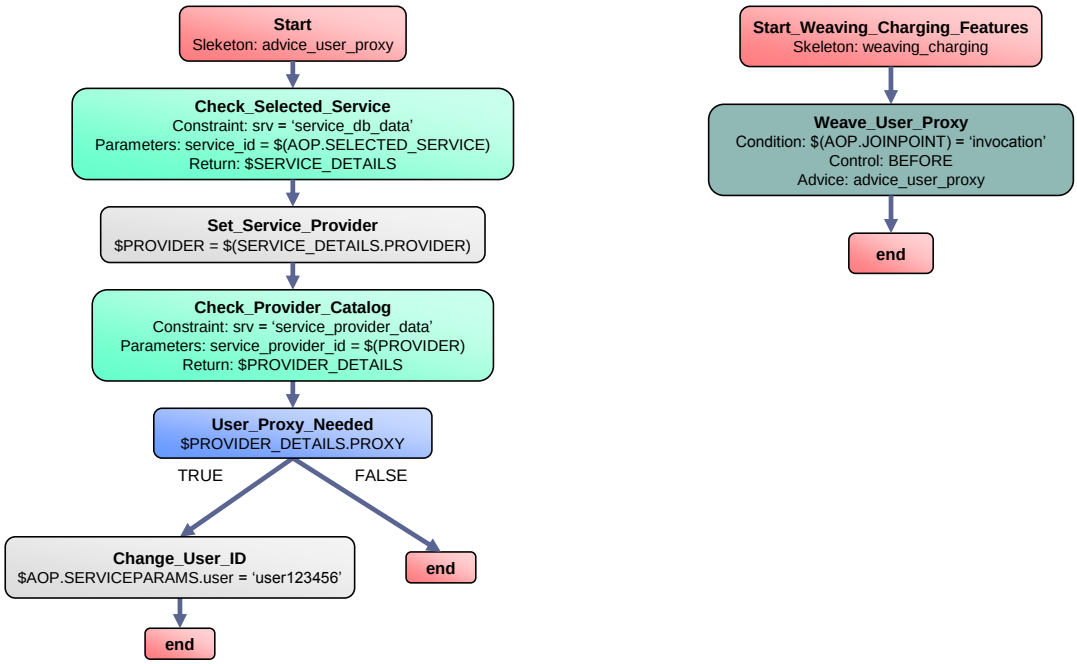

Figure 5.5: The advice and weaving skeletons for user ID replacement

service provider. With this id the advice queries the service provider catalog in order to see if services of this provider require to be run under proxy user credentials. If yes, the advice overwrites the service invocation parameter of the service template with the service provider's ID string. This advice needs to be executed after the selected service is determined and before it is actually invoked. This corresponds to the weaving point before the service invocation join-point. Figure 5.5 shows the used weaving instruction and advice skeletons.

The example here is developed having the localized weather forecast service from Figure 2.18 in mind. The actual weather forecast is in this respect considered to be a 3rd party service that shall not be charged to the end-user. The aspect implementation is however not specific to this example. The user ID replacement is solely dependent on the service database. The advice generically follows this data. The weaving in the composition engine is by default global, which makes the advice applicable to all composite applications. This means, whenever a 3rd party service is used that shall not bill the end-user directly according to the service database, the proxy user is used instead. This way it would not be necessary to update the aspect every time a new 3rd party service is made available.

This example demonstrates that also more complex logic can easily be used in advice. It verifies that not only trivial use cases can be implemented. Especially the use of external services from the advice proofs to be highly useful for creating versatile and generic aspects. It allows to reduce the dependency of aspects to specific composite applications by using external data and logic.

5.1.4 Example: Selecting the Cheapest Service

Using the aspect described in Chapter 5.1.3 the constituent service usage is charged to the provider of the composite application, while the application provider charges the user for the usage of the composite application as a whole. The gain for the application provider is difference between revenue generated by the user and the spendings for constituent services. In this situation it is in the interest of the service provider to prefer the cheapest constituent services [87]. This requirement would need a changed service selection algorithm. It should select that constituent service candidate that charges the best price for the needed service. This example demonstrates how this change of service selection logic can be achieved using aspects.

The left skeleton of Figure 5.6 is the used advice. It is weaved using the weaving instruction on the right side of Figure 5.6. The weaving skeleton includes also the weaving instruction for replacing the charged user as explained in Chapter 5.1.3. This demonstrates how the weaving for several related concerns can be combined into a single skeleton for simple concern management. In this example the weaving skeleton constitutes a module of charging related aspects.

The advice is weaved into the service template before service invocation. At this point a list of service candidates is available. The advice requests the latest price for each of the constituent service candidates and sorts the list with the cheapest service on top. The idea is that the default selection algorithm will always selects the first in the list, thus, placing the cheapest service fist will select it.

In this example the advice has not directly changed anything in the skeleton logic or shared state data of the composite application. The applied changes complement the service selection algorithm, and therefore integral logic of the composition engine. This is effectively a partial re-programming of the execution environment rather than the application. Although there are limited possibilities to do so, but direct influence on the execution engine is usually not done in AOP. This is a useful possibility enabled by the exposure of select internal run time data from the composition execution.

The two weaving instructions put advice into the same join-point. In this case both advices are weaved in before the service invocation. Both advices also directly interact with service selection information. This means that there is a clear advice interaction situation where the actions of one advice influence the other. In this case it is beneficial to first do the service selection optimization for cheapest service. Once this has finally found the preferred service candidate, the change of user ID for referral of charging can be applied. The order of weaving instructions in the weaving skeleton determines the order in which the advice is applied.

If the weaving of interfering aspects is spread over several weaving skeletons, the skeleton priority parameter determines the order in which weaving conditions are evaluated, and therefore also the order of advice execution. This example verifies the need to solve aspect interaction situations. It outlines two combined mechanisms for controlling interaction by advice execution order. It can be done by means of skeleton priority and order of weaving instructions within a skeleton.

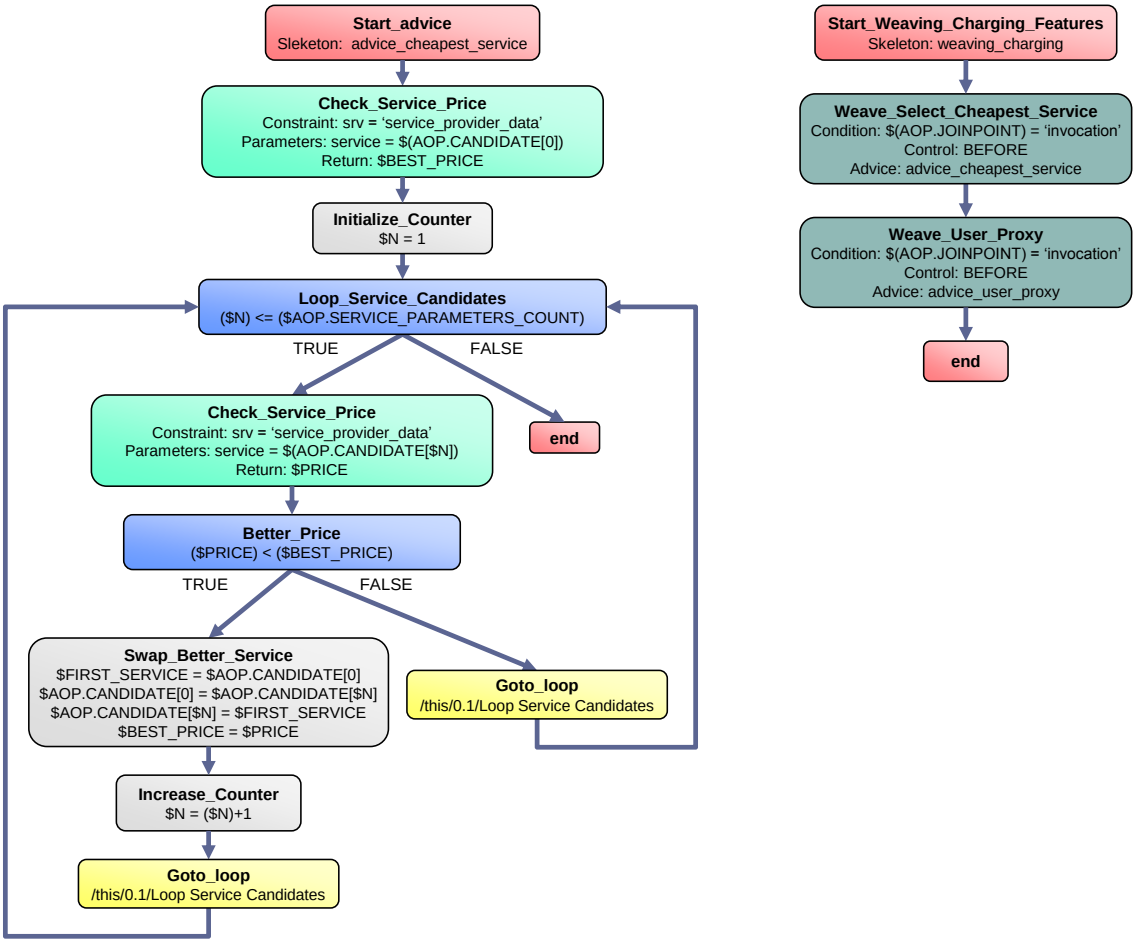

Figure 5.6: The advice that determines if the user ID shall be replaced for charging purposes

5.1.5 Example: New Communication Method Added

The weather service example from Figure 2.18 in Chapter 2.2.12 knows two ways of communicating the service result back to the user: SMS and email. A variant of this service shall be able to use a third way: playing a voice message. This means, the SIP invite that has been used to invoke the weather service would then be routed to an IVR that reads the weather forecast to the user.

This scenario bears a couple of challenges if aspects shall be used to reach this behavior: Another communication method would usually lead to another branch in the skeleton execution. This is a structural change that the proposed AOP techniques does not support directly.

Another challenge is the need for a SIP service being instantiated from within an advice in order to play the weather forecast. Therefore, also the reaction on subsequent SIP messaging would be within the advice. Furthermore, the result of the weather forecast needs to be communicated to the IVR to be played to the user. Also, a web service is used for generating the voice message that is then played to the user using the SIP service. This

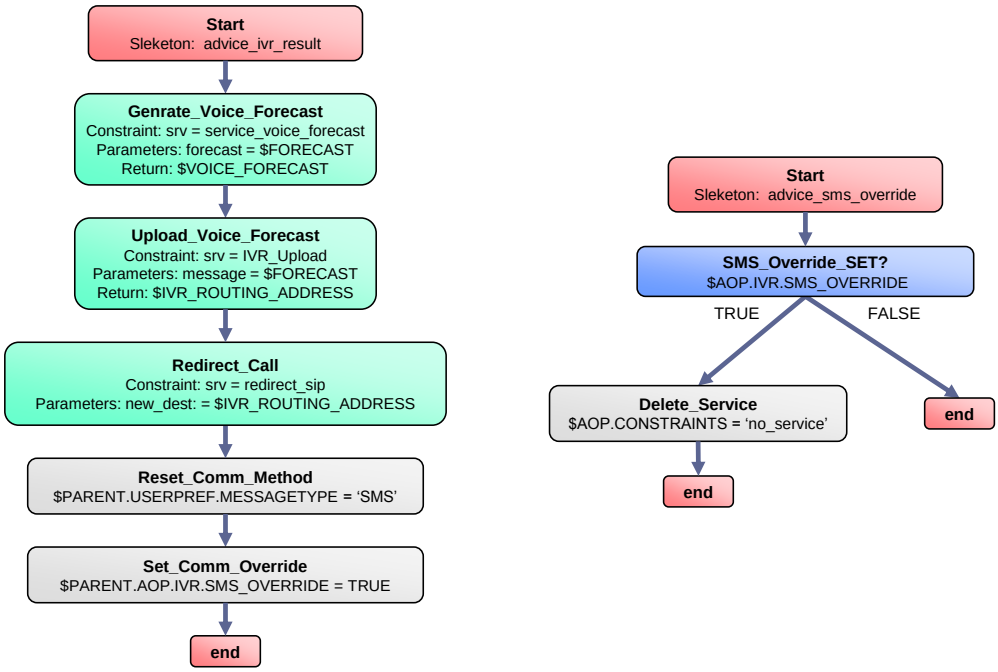

Figure 5.7: The advices that act on the IVR being selected by the user as communication method and override the SMS

means the example operates with heterogeneous services.

The two advices shown in Figure 5.7 work together in order to play the weather forecast as voice message. Figure 5.8 shows the weaving instructions for both advices. The left advice in Figure 5.7 is called "advice_ivr_result". It is weaved into the base skeleton execution after the service that retrieves user preferences. The weaving instruction contains the condition to only weave in this advice is the preferred message type is "VOICE". The advice is therefore skipped for any other preferences and the skeleton would be executed as usual.

If the preferred message type is voice and the advice is executed, it first generates a voice message to be played based on the weather forecast data in the shared state. The voice message itself or a reference to the voice message file stored in shared state. A generic interactive voice recognition (IVR) service might be used to play the message. This would be a SIP service that is selected and instantiated from the advice. First the voice message is uploaded to the IVR system. In order to initiate that it is played, the call is redirected using the "redirect_sip" service with modified destination address towards the IVR. This implicitly initiates a continuation of the SIP call setup. The composition session is halted while the sip call is routed to the IVR and the message is played.

The composition is continued on reception of a subsequent message. This can, for example, be the release of the call after the user has finished listening to the weather forecast message. In this case the execution is resumed within the advice. The next steps are to change the preferred message type to SMS and to write a flag into the shared state

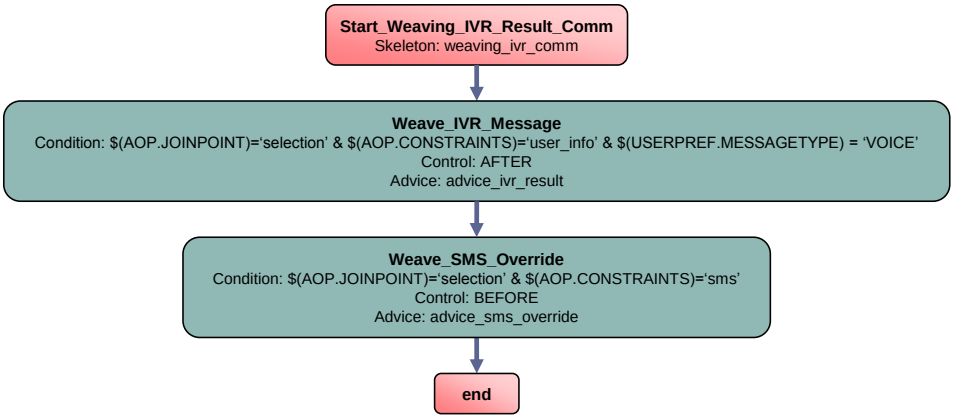

Figure 5.8: The weaving instruction that is used for using IVR as communication method

that indicates that the SMS service shall not be executed. The preferred message type is set to SMS in order to avoid that the following branching condition in the base skeleton selects the SMS branch.

The purpose of the advice "advice_sms_override" in Figure 5.7 is to inhibit the sending of the SMS message if the flag is set to override SMS. It is doing this by overwriting the service selection constraint of the service template that would usually initiate the SMS sending service. This new constraint string would be instantiated with a dummy service, that takes no actions, and therefore, no SMS is sent.

In this example weaving instruction rather than advice logic evaluates the user preference. This demonstrates selective weaving based on dynamic run time data and verifies its usefulness. This example also demonstrates inter-work and communication of two advices through session shared state.

The execution of SIP services and interaction with SIP sessions works seamlessly also from within aspects. This is a consequence of minimizing the difference between skeleton execution of base skeletons and advice. The only difference is the shared state session handling and that is mostly transparent to service invocation. This example verifies, that one of the main properties of the Ericsson Composition Engine, the operation with heterogeneous services, is preserved with the AOP solution.

5.2 Performance Measurements

AOP based on online weaving implies life monitoring of the application execution. The AOP weaving engine continuously observes the all composition execution in search for join-points. This is done even if no advice is applied. It is executed on top of the business logic of a base application and the composition execution. Therefore, an AOP enabled composition execution engine using online weaving naturally implies a server load and application performance penalty even without aspects being actually used.

The most important performance indicator of composite applications is the overall execution latency. It is the time until the application answers a user request. In order to explore this latency the performance of a composite application implemented using AOP is compared with the performance that can be reached using traditional, non-AOP implementation methodology for the same concern. Within the scope of this thesis, traditional implementation refers to direct edits in the base application's skeleton.

The impact on application execution latency means a direct impact on real time capabilities. This is a critical property especially for telecommunication applications. For many use cases involving telecommunication sessions only a small capacity and latency budget would be available in return for the flexibility gained through AOP. It is therefore essential to control and predict the performance figures once aspects are used.

The goal of the performance measurements is not only getting an idea about how well the prototype implementation is fulfilling use case requirements. It is even more important to quantify the extent to which a part of the aspect weaving process is contributing to the overall performance. This will provide valuable insights that help evaluating individual design decisions of the proposed AOP concept.

All measurements are done using a prototype implementation of AOP concepts as proposed in Chapter 4. It was implemented as addition to the Ericsson Composition Engine.

5.2.1 Measurement Method and Key Parameters

The software environment used in the measurements consists of a Sailfin version 1.0 SIP application server based on Glassfish version 1.5 JEE server. It implements the SIP servlet API 1.1. The service database is hosted by an OpenDS version 2.2.1 LDAP database. In this environment an AOP enhanced version of the Ericsson Composition Engine is used. It is based on the research prototype of the Ericsson Composition Engine and not on the Ericsson product of the same name.

The hardware for execution the measurements is a Lenovo laptop computer equipped with Intel Core i7-2620M processor, 8 GB RAM and a solid state disk. The operation system is Windows 7 professional 64 bit. This is not a production grade server platform, but rather an advanced office machine. The absolute values from the measurements therefore do not represent the same performance that can be expected from an optimized application server. This is not a big limitation, because it is mainly the relative comparison between AOP based and non-AOP based implementations that is of interest.

In all measurements the execution time needed by the composition engine for executing a given composite application is measured. For doing so a new composition execution agent was developed. This agent invokes the composition core for execution of a test skeleton. Figure 5.9 shows the measurement environment with the composition engine deployed on the JEE application server together with the measurement CEA and used constituent services. The service database is co-deployed on the same physical machine. Management of services and skeletons is done using the Service Creation Environment. Setup and start of the measurements is controlled through the Eclipse IDE.

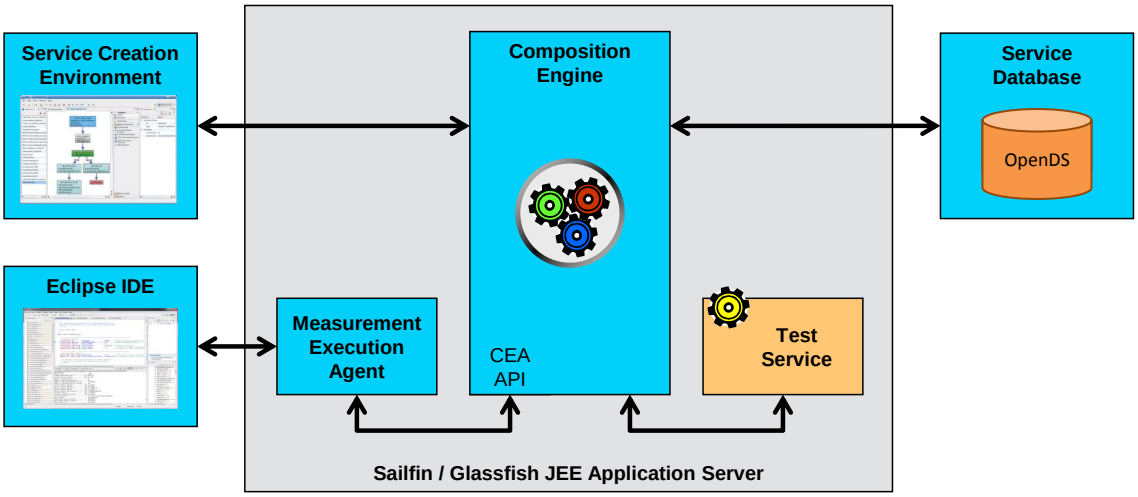

Figure 5.9: Components of the measurement environment

The measurement CEA takes a time-stamp directly before invoking the composition engine, and thus, directly before the skeleton is executed. Another time-stamp is taken directly after the composition engine reports back that the execution has finished. The difference between the two timestamps is the measured overall execution time. This measurement is automatically repeated many times in order to create a data set for statistical evaluation. The measurement CEA directly calculates basic statistical properties, such as minimum, average and median values for the execution time dataset. Furthermore, it generates a distribution of measurement values for creating a histogram together with standard mean and standard deviation.

For measuring with AOP, the measurement CEA can be configured to load weaving instructions prior to starting the measurement. For measuring AOP there are therefore at least three skeletons needed: the base composite service, the skeleton that contains the weaving instructions and the advice that is applied.

Figure 5.10 shows a generic base skeleton that is used in most performance measurements. It implements an execution loop with L being the maximum number of loops. Skeleton elements to be evaluated in the measurement are placed inside the loop. This could, for example, be a service template. It is then executed L times during a single execution of the composite application. Executing L loops also means that advice applied to the element in the loop is also executed L times per execution of the base composite service. The measurement CEA executes the entire composite application many times and in each of these executions the loop is executed L times.

There is also a special constituent service that is frequently used. This service is called aop_test. It is implemented internally within the composition engine. This means that its invocation is very fast compared to external methods, such as SOAP for web service invocations. The aop_test service does not implement any function and it does not execute any logic. It exits immediately after being invoked. This also means, that it offers only

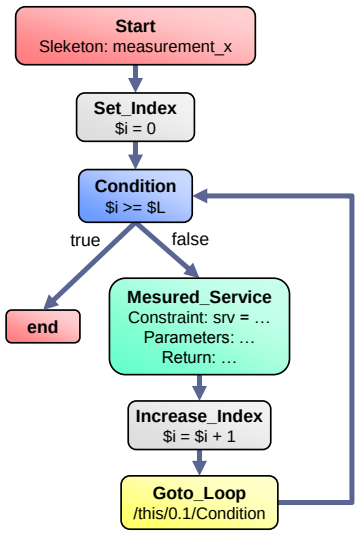

Figure 5.10: Basic test skeleton

a simple API. It does not contain any invocation parameters and it does also not provide data in its reply. It is therefore a very simple and fast to execute dummy service.

As a first check of the measurement system and to get an idea of the quality of results, the execution of the base skeleton from Figure 5.10 was measured without any aspects being applied. The result is shown in Figure 5.11. It shows the measured skeleton execution times for each of the 100000 measurements that were performed. Here the results are shown for 1, 5 ,10 and 15 loop iterations in the test skeleton.

Clearly visible in Figure 5.11 is the high scattering of the measured execution time. There are naturally many other applications and services running on the measurement machine next to the composition execution environment. Also, the underlying Java run time environment regularly performs actions, such as garbage collection, that create processing load. This causes an unavoidable and unpredictable disturbance to the composition execution and consequently also to the measurement.

Because of the constant execution disturbances, it is important to execute as simple examples as possible in order to keep the time short, that is spent on something else than skeleton and aspect execution related activities of the engine. Composite applications with long execution times are impacted by potentially many disturbances. If the execution finishes fast, there is a good chance to find a good number of instances with minimal disturbances. This can also be seen in Figure 5.11. For higher numbers of loop iterations the execution time is not only bigger, but it is also more scattered by accumulated disturbances.

The focus of this evaluation is not on absolute server performance, but on the performance characteristics of the composition engine implementation. The important property for this consideration is the execution under optimal circumstances. This means execution sessions that are least impacted by disturbances are most interesting.

Figure 5.11 shows a sharp lower edge of each dataset. There is a clear minimum

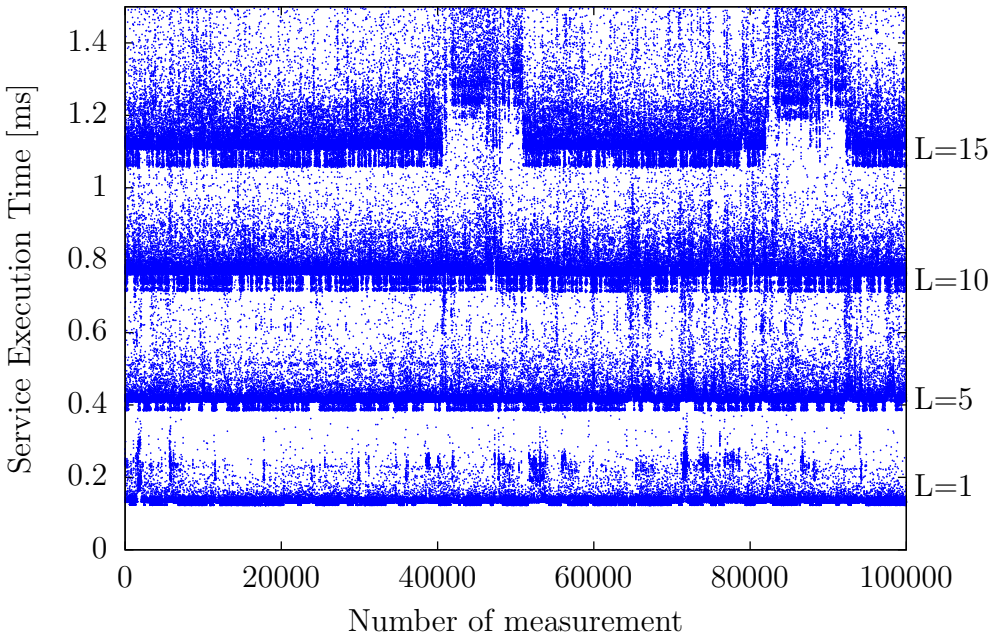

Figure 5.11: Measurement of the base service without using AOP

execution time. This can be interpreted as the time the composition engine needs at least for the skeleton execution if it is not disturbed by other processes and parallel services. This minimum value is therefore the key measurement result. To the greatest possible extend only the composition engine code contributed to the execution time visible in the minimum time.

For this reason a high number of measurement samples determine the minimum execution time boundary that is possible for the measured scenario. Figure 5.12 shows the minimum value and the median for each of the measurement sets. With good accuracy the measured minimum shows a linear increase of execution time over higher numbers of loop iterations. Each additional loop iteration takes about 0.0625 ms. The median values are also linearly dependent on the loop iterations, but there is a clearly visible scattering of measured results that increases with absolute execution time.

The influence of disturbances becomes particularly visible in the histogram that shows the distribution of measured execution times. It is shown in Figure 5.14 for the measurement of the base skeleton with $L = 10$ iterations. The minimum, median and average of the execution times are:

$$T_{minimum} = 0.710\,\text{ms}$$
$$T_{median} = 0.770\,\text{ms}$$
$$T_{average} = 0.817\,\text{ms}$$

The standard deviation expresses the dispersion from the average. It is a measure of the scattering in the results caused by disturbances. For the measurement with L=10 loop

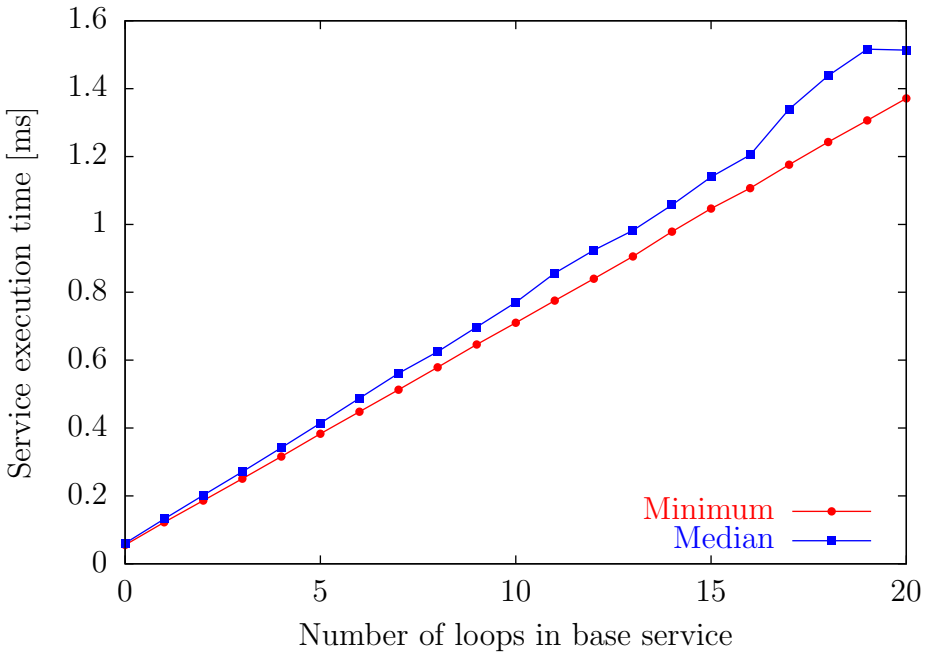

Figure 5.12: Minimum and median values

iterations it is $\sigma_{10} = 0.248\,\text{ms}$. The histogram in Figure 5.14 shows three distinct peeks. This indicates a regular disturbance of the composition execution that adds a specific execution latency. What exactly caused these disturbances was not further investigated. It does not influence the final evaluation because the minimum execution time is used as most relevant metric. This minimum time is the leftmost edge of the histogram.

Figure 5.13 shows the histograms for measurements with different numbers of loop iterations. The wider graphs for increasing loop iterations shows once more the increasing deviation caused by disturbances.

End-to-end latency varies with the used composition execution agent and the underlying service technology. This is independent of the composition execution performance within the composition engine where also the AOP environment is implemented. Therefore, the choice of composition execution agent does not influence the measurement of the performance penalty incurred by the AOP framework. This independence was briefly verified in a number of test runs with the same skeleton where the service was instantiated by different technologies. In these tests the end-to-end latency varies, but the execution time contribution from composition and AOP stays constant. This means simple constituent services and a simple execution agent can be used for measurements.

This implies that special measurements for SIP services are not necessary. The proposed AOP environment operates entirely in the service technology agnostic part of the composition engine. It will therefore treat the composition of SIP services in no way different from other service technologies. Measuring with SIP based composition invocation and SIP constituent services shows that a major part of the overall execution time is contributed

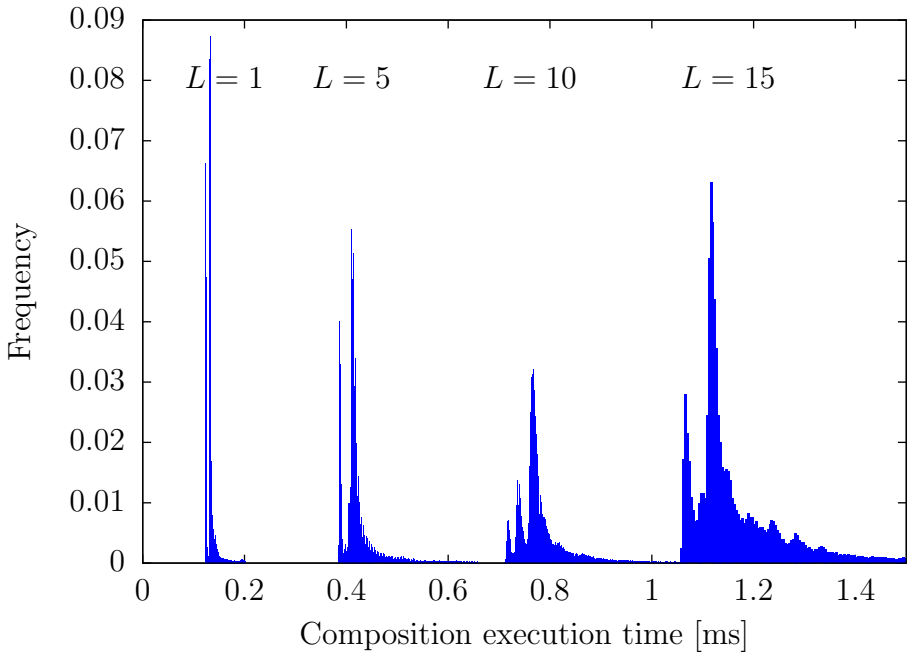

Figure 5.13: Distribution of execution time for 1, 5, 10 and 15 loop iterations

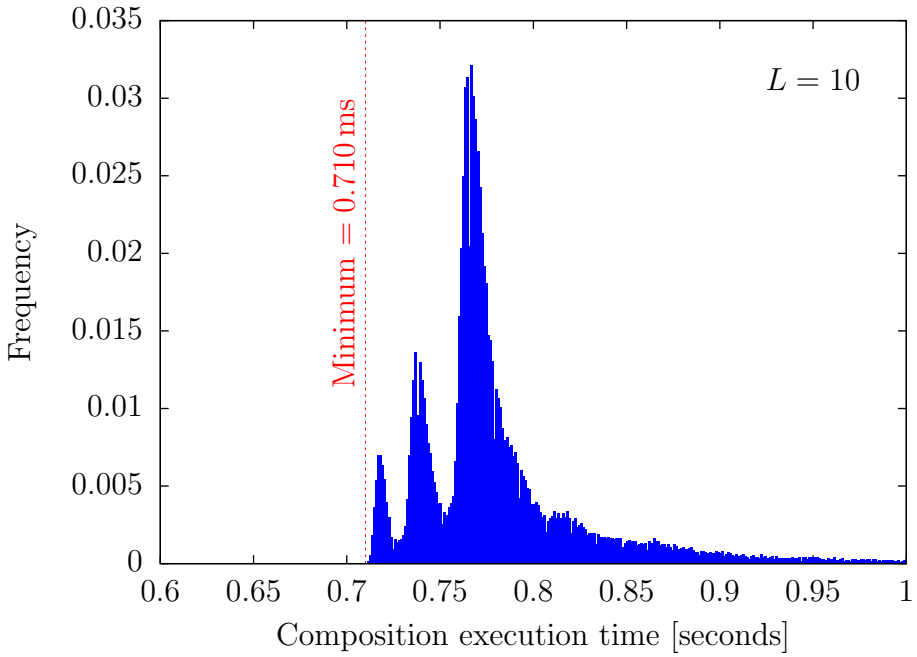

Figure 5.14: Distribution of measured execution time for $L = 10$ loop iterations

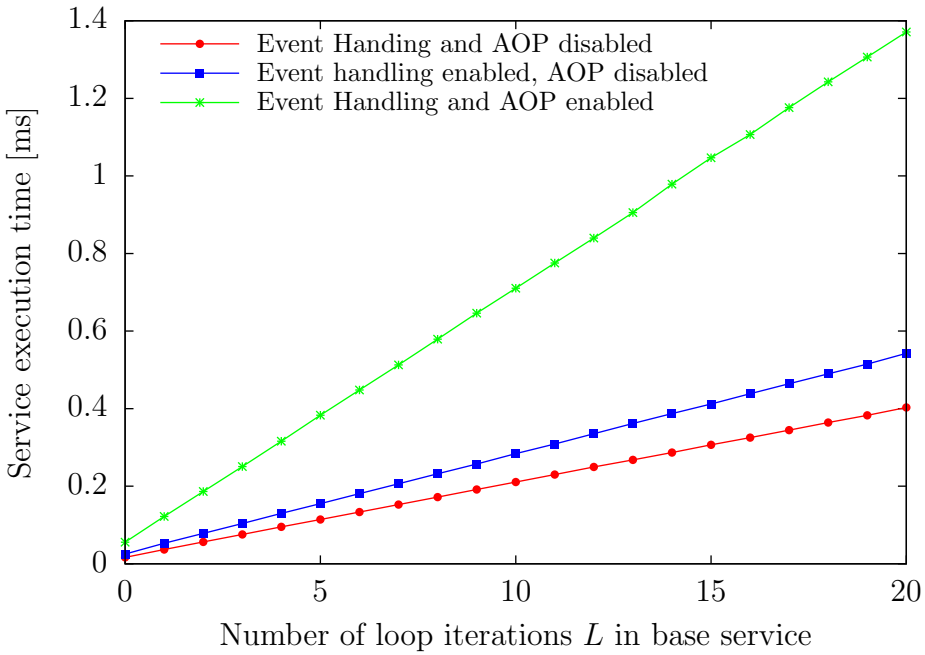

Figure 5.15: Minimum execution times measured with and without event handling and AOP enabled

by processing the SIP stack rather than the composition engine. Previous measurements have verified, that the composition engine is relatively fast compared to the time needed by SIP containers for protocol and service handling. Using SIP would therefore only lead to longer execution times with more disturbances in the measurements without any additional insights for evaluating the AOP implementation. Nevertheless, the measurement CEA uses the same API methods as the SIP CEA for invoking the composition core.

5.2.2 Latency Contributed by Event Handing and AOP

The AOP environment is build on top of an event handing infrastructure within the composition engine. In this respect join-points throw events and the weaving engine is a dynamic event router and advice acts as event handler. It is possible to disable the AOP and event handling. This allows to assess the performance of the bare engine, and the impact of event handling and AOP individually.

The same measurement as in Chapter 5.2.1 is executed three times with different configuration settings of the composition execution engine. The result is shown in Figure 5.15. It shows that event handling has a clear impact on the performance of the composition execution. the execution time increases by about 35%. If on top of event handling also AOP is enabled, the execution time is increased by 235%. This shows a major impact from event handling and an even bigger impact from the online weaving implementation.

At the time of implementation of the AOP framework, AOP is the single major user of

Event handling. This means that most of the events that are currently generated by the composition engine are related to join-points. An event is thrown at each weaving-point. This means, with the activation of AOP, event handling is getting busy with join-point events.

This measurement allows to determine some characteristics of composition execution. The execution time of the base skeleton T_{base} depends mainly on the number of iterations n in the skeleton loop. Each loop execution adds T_{loop} to the execution time. The total execution time of this basic measurement skeleton is therefore calculated as:

$$T_{total}(n) = T_{base}(n) = T_{loop}\, n + T_{other} \tag{5.1}$$

where T_{other} is the time needed for execution of the skeleton elements outside the loop.

The values for T_{loop} and T_{other} can be determined from the measurement of T_1 and T_2 that correspond to measurement parameters $n = L_1$ and $n = L_2$:

$$T_{base}(n = L_1) = L_1 T_{loop} + T_{other} = T_1$$
$$T_{base}(n = L_2) = L_2 T_{loop} + T_{other} = T_2$$

This leads to:

$$T_{loop} = \frac{T_2 - T_1}{L_2 - L_1}$$
$$T_{other} = T_1 - L_1 T_{loop} = T_2 - L_2 T_{loop}$$

For $L_1 = 1$ and $L_2 = 10$ the measured values are $T_1 = 0.122\,\mathrm{s}$ and $T_{10} = 0.710\,\mathrm{s}$, thus:

$$T_{loop} = 0.0654\,\mathrm{ms}$$
$$T_{other} = 0.0568\,\mathrm{ms}$$

This means that each iteration of the skeleton loop takes $T_{loop} = 0.0654\,\mathrm{ms}$

5.2.3 Latency Introduced by Weaving Execution

The presence of a weaving engine and the execution of weaving instructions has an impact on the time needed for execution of a composite application. This refers to the impact of just evaluating the weaving instructions, even if none of them is actually met, and thus, no advice is executed. The effort for weaving obviously increases with the number of weaving instructions that need to be checked. This impact the number of loaded weaving instructions has on the execution time of a composite application is explored in this chapter.

In order to explore the impacts of weaving, the basic skeleton is used as shown on the left in Figure 5.16. This means, a simple composite application with a service template in

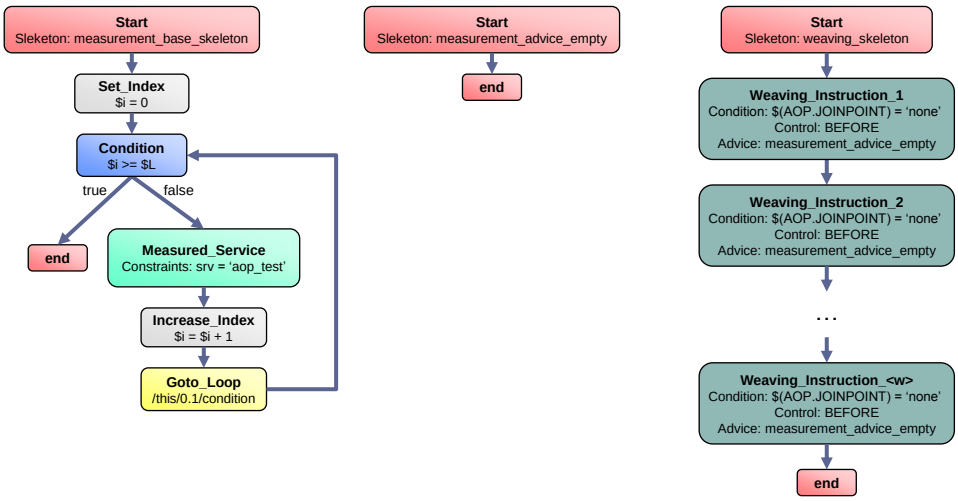

Figure 5.16: Skeleton, advice and weaving instructions measuring of weaving latency

a loop is executed. Weaving instructions are loaded using the weaving skeleton as shown on the right of Figure 5.16. Its weaving condition matches a join-point name that does not exist, thus, the condition is never met. There is also an simple advice skeleton defined, but in this measurement the advice is never invoked.

For the measurement the number of loaded weaving instructions is varied between $I = 0$ and $I = 100$. For each weaving setup the composite service is executed 10000 times and the time needed for each execution is measured. Furthermore the number of iterations of the loop in the base skeleton is also varied between $L = 0$ and $L = 10$. The result of each measurement is the minimum execution time that could be reached for a given number of loaded weaving instructions I and skeleton loop iterations L.

The number of weaving instructions is varied in order to explore if it has an impact on the execution time of the composition. With the number of skeleton iterations varies the number of join-points that are present in the skeleton. With the AOP implementation used for measurements, the basic skeleton contains five join-points and a total of 10 weaving points. The weaver checks all loaded weaving instructions at each weaving point in order to determine the advice that needs to be started.

The measurement results shows, that the processing effort for checking weaving conditions is huge, resulting in a considerable increase service execution time. The results presented in Figure 5.18 show the measured minimum service execution times for varying number of loaded weaving instructions. Figure 5.17 shows the minimum execution time when varying the number of processed join-points J and the related number of weaving points W by means of different numbers of skeleton loop iterations L. Especially the number of weaving points is important, because at each weaving point all weaving instructions are checked.

A theoretical analysis helps to understand the quantitative dependencies: $W_{element}$ is the number of weaving-points being executed per skeleton element type. The number

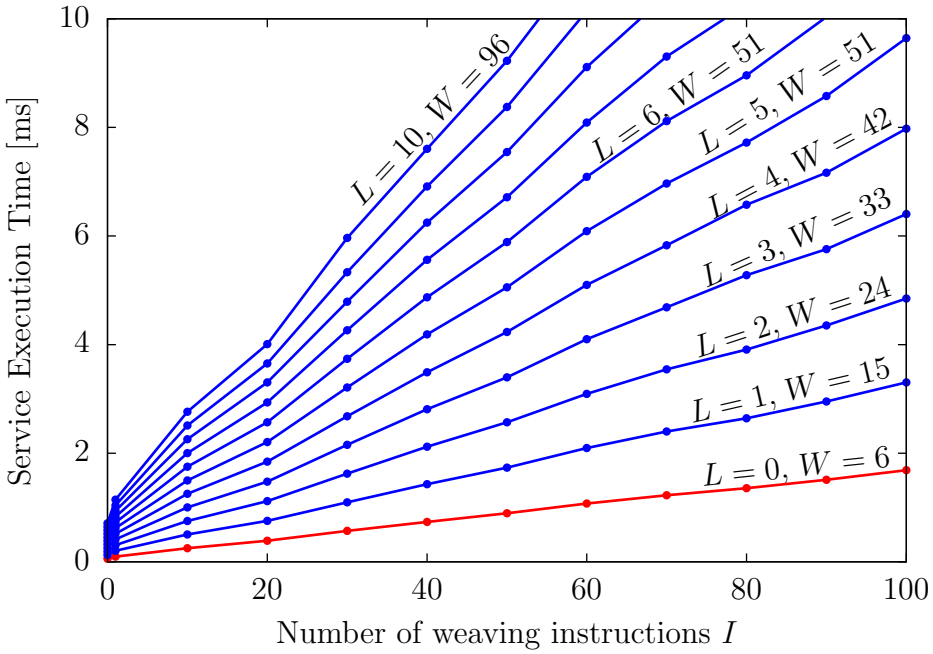

Figure 5.17: Service execution time increase because of weaving. Dependency on number of loaded weaving instructions

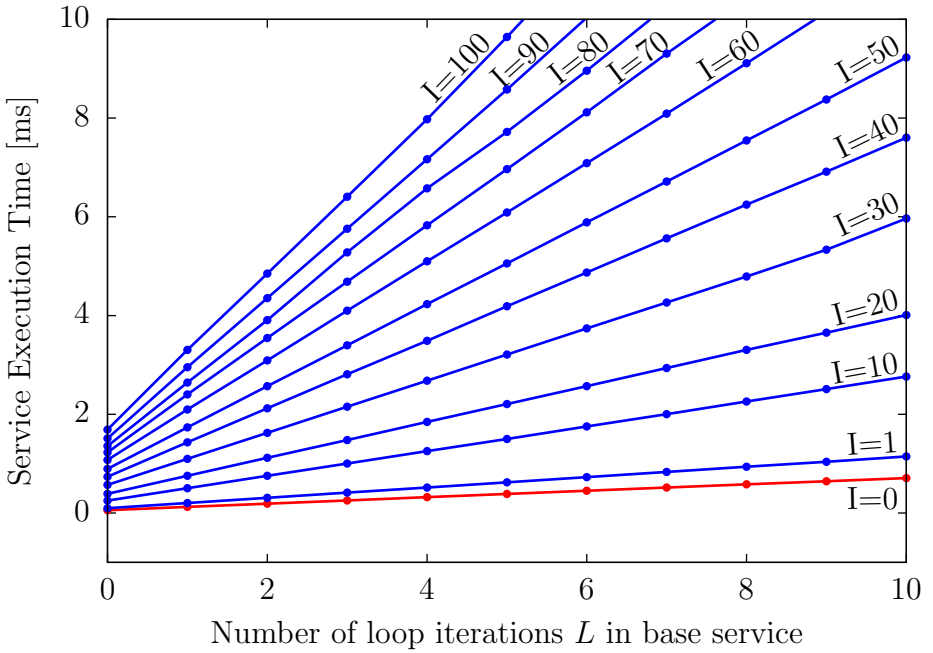

Figure 5.18: Service execution time increase because of weaving. Dependency on number of join-points

Skeleton Element	Join-Point	Weaving-Points	Number of Weaving Points	
			Proposed	Implemented
Start Element	start	after	$W_{start} = 1$	$W_{start} = 1$
End Element	end	before	$W_{end} = 1$	$W_{end} = 1$
Service Template	template selection invocation	before, after, around before, after, around before, after, around	$W_{template} = 9$	$W_{template} = 5$
SSM Command	ssm	before, after, around	$W_{ssm} = 3$	$W_{ssm} = 2$
Condition Element	condition	before, after	$W_{condition} = 2$	$W_{condition} = 2$
Goto Element	goto	before, after, around	$W_{goto} = 3$	$W_{goto} = 0$

Table 5.1: Number of weaving-points per skeleton element type

of weaving-points per element type depends on the number of weaving points that are associated with the element. Table 5.1 shows per skeleton element type, and therefore per related join-point how many weaving-points are executed. Not all weaving-points that were proposed in the concept design are actually implemented. Table 5.1 therefore distinguishes the proposed number and the implemented number of weaving points.

For each skeleton element the number of weaving-point executions is the number of weaving points associated with the skeleton element $W_{element}$ multiplied by the number of times this skeleton element type is executed $E_{element}$. The total number of executed weaving-points W_{total} is the sum of all skeleton element specific weaving-point executions:

$$W_{total} = \sum_{\text{All Skeleton Elements}} E_{element}\, W_{element} \tag{5.2}$$

Please note that $E_{element}$ expresses the number of executions per skeleton element type and not the number of skeleton elements that are present in the skeleton. Equation (5.2) can be expanded by skeleton element type:

$$\begin{aligned} W_{total} = {} & E_{start}\, W_{start} + E_{end}\, W_{end} + E_{template}\, W_{template} + E_{ssm}\, W_{ssm} + \ldots \\ & \ldots + E_{condition}\, W_{condition} + E_{goto}\, W_{goto} \end{aligned}$$

In general, there can also be multiple start elements, because a skeleton can call sub-skeletons by means of the goto element. With the values from Table 5.1 the following formula expresses the number of weaving points that are executed within the execution of a composite application's skeleton:

$$W_{total} = E_{start} + E_{end} + 5E_{template} + 2E_{ssm} + 2E_{condition} \tag{5.3}$$

Number of Iterations n	Number of Join-Points $J_{total}(n)$	Number of Weaving Points $W_{total}(n)$
0	4	6
1	7	15
2	10	24
3	13	33
...	...	...
n	$4 + 3n$	$6 + 9n$

Table 5.2: Number of join-points and weaving-points in the example service, dependent on number of iterations

At each weaving point the data environment for checking the weaving conditions is created. This action assembles the shared state data from the composition session and combines it with engine internal run time variables that need to be exposed to the weaving engine and advice. This is done once per weaving point. Based on this data the weaving engine checks all conditions of weaving instructions. All weaving instructions are global, and therefore, the entire set of loaded weaving instructions is checked at every weaving-point. Consequently, the total number of weaving checks C_{total} that are executed while executing a composite service is the number of weaving-points multiplied with the number of weaving instructions I:

$$C_{total} = I\, W_{total} = I\ (\ E_{start} + E_{end} + 5E_{template} + 2E_{ssm} + 2E_{condition}\ ) \tag{5.4}$$

The particular skeleton being used in these measurements contains a loop with n iterations. This means each skeleton element in the loop is executed n times and each skeleton element outside the loop is executed once. The total number of executed weaving-points W_{total} is a function of the number of loop iteration n:

$$W_{total}(n) = W_{loop}\ n + W_{other} \tag{5.5}$$

The equation (5.5) distinguishes the skeleton elements being executed within the loop W_{loop} and outside the loop W_{other}. Table 5.1 provides the number of weaving-points per skeleton element and Table 5.2 summarizes the values for J and W for the measured example skeleton dependent on the number of loop iterations.

$$W_{loop} = 2E_{condition} + 5E_{template} + 2E_{ssm} + W_{goto} = 9$$
$$W_{others} = E_{start} + 2E_{ssm} + E_{end} = 6$$

With i being the number of loaded weaving instructions, the number of weaving checks in the measured composite service is therefore a function of n and i:

$$C_{total}(n,i) = i\, W_{total}(n) = i\,(W_{loop}\, n + W_{other}) \\ = i\,(9n+6) \tag{5.6}$$

The total execution time T_{total} for a composite application is the sum of the individual execution times from all skeleton elements. Without considering the contribution from AOP the total execution time is:

$$T_{total} = \sum_{\text{All Skeleton Elements}} E_{element}\, T_{element} = T_{base} \tag{5.7}$$

This time T_{total} can be considered constant as long as all execution parameters, such as the number of loop iterations, stay constant and the same constituent services are selected. In the test environment this is the case. The execution time of constituent services being selected and invoked is included in T_{base}. Disturbing factors from the execution platform are neglected here because the evaluations are based on minimum execution time with minimal contributions from these disturbances.

Not considering AOP means that only the execution time from the composition execution engine while executing the base application skeleton is considered and not the time needed for weaving and advice execution. Within the scope of this measurement and based on the measured skeleton from Figure 5.16, $T_{total}(n)$ and $T_{base}(n)$ are both functions of the skeleton loop parameter n. Under these conditions the equation (5.1) with measured values for T_{loop} and T_{ohter} is applicable here and it determines $T_{base}(n)$.

The total execution time considering aspect weaving and advice execution is the sum of the execution time for the base skeleton T_{base}, plus the time that is needed for all weaving and advice execution:

$$T_{total} = T_{base} + T_{weaving} + T_{advice} \tag{5.8}$$

There are two major actions contributing to the time for weaving: The preparation of data used for condition evaluation and the time for executing the condition check for all loaded weaving instructions. The data preparation consists of creating a copy of the shared state and the addition of AOP specific data. This action is executed at each weaving point and the needed overall execution time is expressed by T_{data}. It is considered to be the same for each weaving-point type because the implementation is very similar.

The data preparation time at each weaving-point multiplied by the number of executed weaving points according to (5.5) provides the total execution time contribution from weaving data preparation $T_{preparation}$. Due to the loop in the used skeleton, $T_{preparation}$ is a function of number of iterations n.

$$T_{preparation} = T_{data}\, W_{total} \tag{5.9}$$

$$T_{preparation}(n) = T_{data}\, W_{total}(n) = T_{data}\,(W_{loop}\, n + W_{other}) = T_{data}\,(9n+6)$$

The effort for checking each single weaving condition is considered to be constant and expressed by the execution time T_{check}. This time multiplied by the total number of executed checks according to (5.6) provides the total contribution to the composite service execution from checking the weaving instructions.

$$T_{conditions} = T_{check}\ C_{total} \tag{5.10}$$

$$T_{conditions}(n,i) = T_{check}\ C_{total}(n,i) = T_{check}\ i\,(W_{loop}\ n + W_{other}) = T_{check}\ i\,(9n+6)$$

Using (5.8) with (5.9) and (5.10) this means for the total execution time of a composite service:

$$\begin{aligned} T_{total} &= T_{base} + T_{preparation} + T_{conditions} + T_{advice} \\ &= T_{base} + T_{data}\ W_{total} + T_{check}\ C_{total} + T_{advice} \end{aligned} \tag{5.11}$$

With these considerations and the measured values it would be possible to determine values for $T_{preparation}$ and $T_{conditions}$ and also for for T_{data} and T_{check}. Together with the value for T_base as determined in Chapter 5.2.2 this would provide estimates for individual impacts of skeleton execution, and the major sub-activities of weaving on the execution time of a composite application:

Two measured values for the total execution time $T_{total} = T_1$ and $T_{total} = T_2$ are used. They were measured with different configurations of the measurement parameters $n = L$ and $i = I$. The parameters L_1 and L_2 are the number of skeleton loop iterations that were executed when measuring T_1 and T_2. The parameters I_1 and I_2 are the corresponding numbers of loaded weaving instructions. These parameters cause W_1 and W_2 weaving points being executed and C_1 and C_2 numbers of weaving instruction checks being done throughout the measurement. Using these parameters in (5.11) and considering that no advice is executed here, the total execution time for the two measurement points setups can be expressed as:

$$\begin{aligned} T_1 &= T_{base,1} + W_1\ T_{data} + C_1\ T_{check} \\ T_2 &= T_{base,2} + W_2\ T_{data} + C_2\ T_{check} \end{aligned}$$

Where $T_{base,1}$ and $T_{base,2}$ refer to the base service execution times without AOP being used as measured in Chapter 5.2.2. This can be solved for T_{data} and T_{check}:

$$T_{check} = \frac{W_2T_1 - W_1T_2 - W_2T_{base,1} + W_1T_{base,2}}{W_2C_1 - W_1C_2} \tag{5.12}$$

$$T_{data} = \frac{C_2T_1 - C_1T_2 - C_2T_{base,1} + C_1T_{base,2}}{W_1C_2 - W_2C_1} \tag{5.13}$$

For the current measurement the constants W_1, W_2, C_1 and C_2 can be calculated from the respective parameters L_1, L_2, I_1 and I_2:

$$\begin{aligned}
n = L_1 = 1, i = I_1 = 1: \quad & W_1 = 9L_1 + 6 && = 15 \text{ weaving points} \\
& C_1 = I_1\,(9L_1 + 6) && = 15 \text{ condition checks} \\
i = L_2 = 10, i = I_2 = 50: \quad & W_2 = 9L_2 + 6 && = 96 \text{ weaving points} \\
& C_2 = I_2\,(9L_2 + 6) && = 4800 \text{ condition checks}
\end{aligned}$$

The measurement of this chapter shows the following results:

$$\begin{aligned}
T_1 = T_{total}(n = 1, i = 1) &= 0.203177\,\text{ms} \\
T_2 = T_{total}(n - 10, i = 50) &= 9.226700\,\text{ms}
\end{aligned}$$

Using (5.12) and (5.13) the execution time for each shared state data preparation prior to weaving checks is $T_{data} = 0.003703\,\text{ms}$. Furthermore, the execution time needed for checking a single weaving condition is $T_{check} = 0.001700\,\text{ms}$.

These results allow an analysis of the contributions to the composition execution time. In the measured execution for $n = 10$ skeleton loop iteration and $i = 50$ loaded weaving elements, there are $J_{total} = 3n + 4 = 34$ join-points and $W_{total} = 9n + 6 = 96$ weaving-points processed. With $i = 50$ weaving instructions being loaded there are a total of $C_{total} = 4800$ weaving condition checks executed while processing the composite application. The measured total execution time is:

$$\begin{aligned}
T_{total} &= T_{base} + T_{preparation} + T_{conditions} \\
&= T_{base} + W_{total}\,T_{data} + C_{total}\,T_{check} = 9.2267\,\text{ms}
\end{aligned}$$

With T_{base} as measured in Chapter 5.2.2 the total execution time consists of the following contributions:

T_{base}		$= 0.710\,\text{ms}$	7.7 %
$T_{preparation}$	$= W_{total}\,T_{data}$	$= 0.355\,\text{ms}$	3.6 %
$T_{conditions}$	$= C_{total}\,T_{check}$	$= 8.161\,\text{ms}$	88.4%

Some more example results are shown in Table 5.3.

The formula that expresses the total execution time for this measurement as a function of the measurement parameters $i = I$ and $n = L$ is

$$\begin{aligned}
T_{total}(n, i) &= T_{base}(n) + T_{preparation}(n) + T_{conditions}(n, i) && (5.14) \\
&= T_{base}(n) + 9T_{check}\,n\,i + 9T_{data}\,n + 6T_{check}\,i + 6T_{data} && (5.15)
\end{aligned}$$

Parameters	$L = 1$, $I = 1$		$L = 10$, $I = 50$	
Base skeleton execution: T_{base}	0.122 ms	60.1 %	0.710 ms	7.7 %
Weaving execution:				
Join-Points J_{total}	7		34	
Weaving-Points W_{total}	15		96	
Weaving Checks C_{total}	15		4800	
$T_{preparation} = W_{total}\ T_{data}$	0.056 ms	27.3 %	0.355 ms	3.9 %
$T_{conditions} = C_{total}\ T_{check}$	0.026 ms	12.6 %	8.161 ms	88.4 %
$T_{weaving}$	0.081 ms	39.9 %	8.516 ms	92.3 %
T_{total}	0.203 ms		9.227 ms	

Parameters	$L = 1$, $I = 10$		$L = 10$, $I = 1$	
Base skeleton execution: T_{base}	0.122 ms	28.2 %	0.710 ms	57.8 %
Weaving execution:				
Join-Points J_{total}	7		34	
Weaving-Points W_{total}	15		96	
Weaving Checks C_{total}	150		96	
$T_{preparation} = W_{total}\ T_{data}$	0.056 ms	12.8 %	0.355 ms	28.9 %
$T_{conditions} = C_{total}\ T_{check}$	0.255 ms	58.9 %	0.163 ms	13.3 %
$T_{weaving}$	0.311 ms	71.8 %	0.519 ms	42.2 %
T_{total}	0.433 ms		1.229 ms	

Table 5.3: Measurement results and contributions to the overall execution time

with:

$$T_{base}(n) = T_{loop}\ n + T_{other} \tag{5.16}$$

$$T_{preparation}(n) = (9n + 6)\ T_{data} \tag{5.17}$$

$$T_{conditions}(n, i) = i(9n + 6)\ T_{check} \tag{5.18}$$

For a constant $n = L$ equation (5.14) can be simplified to:

$$T_{total}(i) = (9LT_{check} + 6T_{check})\ \ i + (9LT_{data} + 6T_{data} + T_{loop}L + T_{other}) \tag{5.19}$$

This means that the execution time shows a linear dependency of the number of loaded weaving instructions i when the number of loop iterations is constant. This is in line with the measured results as presented in Figure 5.17. The graph is to a great extent approximately linear. The values of T_{loop} and T_{other} where already determined in Chapter 5.2.2 where the same base skeleton was used.

For a constant $i = I$ the following simplified equation can be found for (5.14). It expresses the total execution time as a function of the number of skeleton loop iterations:

$$T_{total}(n) = (9IT_{check} + 9T_{data} + T_{loop})\ \ n + (6IT_{check} + 6T_{data} + T_{other}) \tag{5.20}$$

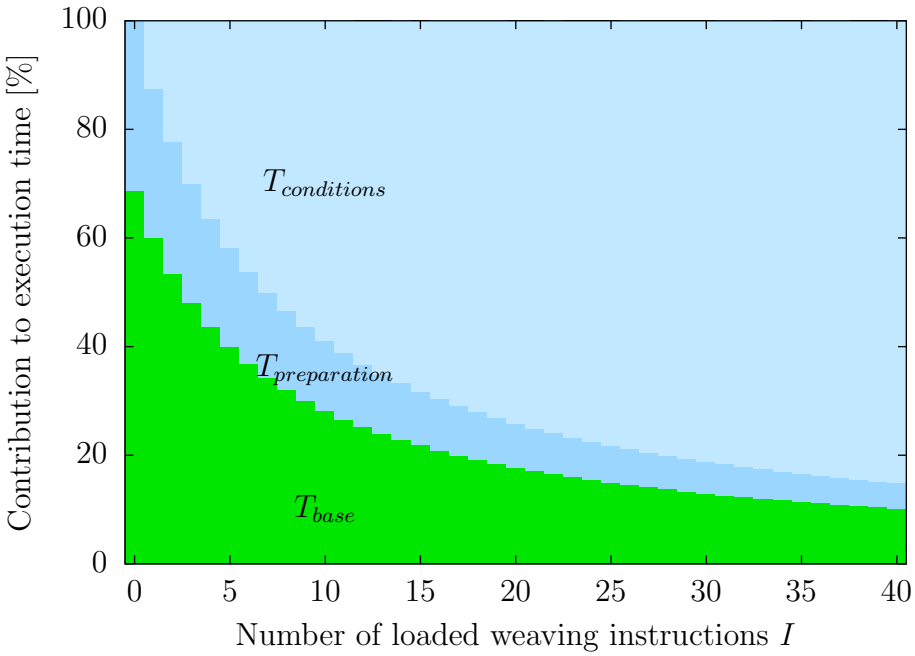

Figure 5.19: Execution time contributions depending on loaded weaving instructions I with constant $L = 1$

The result is a linear dependency of the total execution time from the number of loop iterations. This is also visible in the measurements as shown in Figure 5.18.

These measurements show that a major part of the composite application execution time is due to executing AOP related actions. With only a few weaving instructions are loaded, the execution time spent on AOP already exceeds the time for doing basic composition. This is shown in Figure 5.19. The ratio between the time needed for data preparation and weaving condition checks depends on the number of loaded weaving instructions. For only a small number of weaving instructions the time for data preparation is bigger than the time needed for performing weaving checks. In the example with 50 weaving instructions, checking the weaving conditions needs 88.4 % of the execution time while the data preparation only takes 3.9 %. The effort for event handling and for the weaving engine being the event handler is not distinguished.

The share of a partial execution time of the entire execution time is expressed by time ratio R_{time}

$$R_{time} = \frac{T_{part}}{T_{total}} \tag{5.21}$$

Here T_{part} can be any of the contributing times, for example T_{base}, $T_{preparation}$ or $T_{conditions}$. This share for the example measurement depends on the parameters $n = L$ and $i = I$. Combining (5.14) with (5.1) provides the total execution time $T_{total}(n, i)$ that is valid for the example skeleton of this measurement:

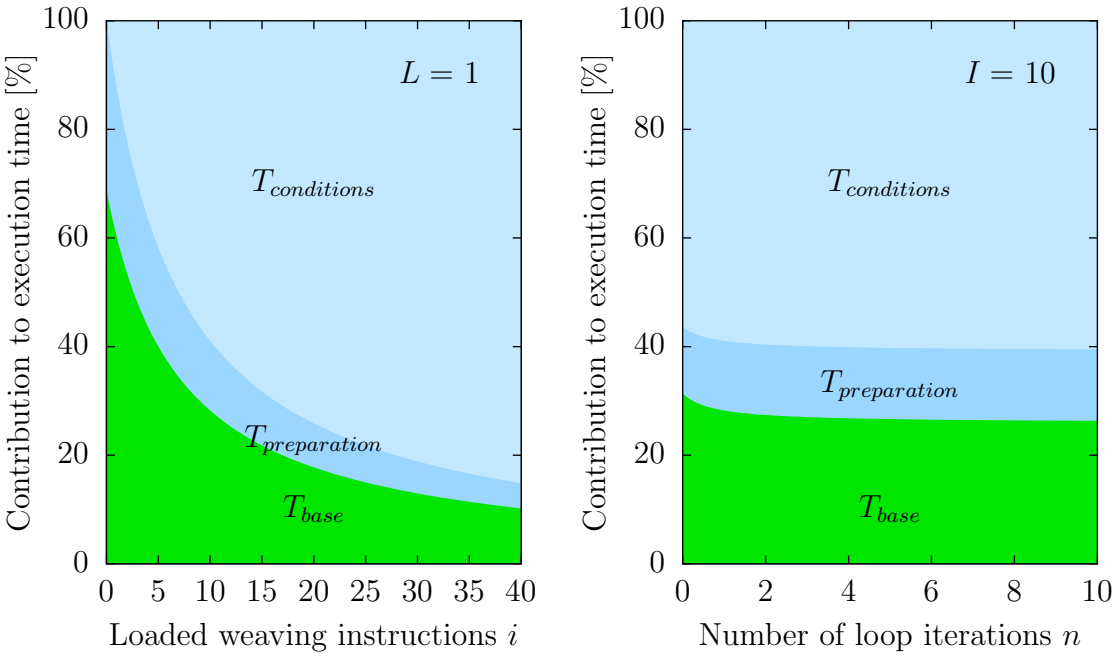

Figure 5.20: Execution time contributions based on theoretical considerations

$$T_{total}(n,i) = 9T_{check}\ n\ i + (T_{loop} + 9T_{data})\ n + 6T_{check}\ i + (T_{other} + 6T_{data}) \tag{5.22}$$

using (5.16) the share in execution time for the base skeleton execution, shared state data preparation and weaving checks is:

$$R_{time,base}(n,i) = \frac{T_{loop}\ n + T_{other}}{T_{loop}\ n + T_{other} + 9T_{check}\ n\ i + 9T_{data}\ n + 6T_{check}\ i + 6T_{data}} \tag{5.23}$$

$$R_{time,preparation}(n,i) = \frac{(9n+6)\ T_{data}}{T_{loop}\ n + T_{other} + 9T_{check}\ n\ i + 9T_{data}\ n + 6T_{check}\ i + 6T_{data}} \tag{5.24}$$

$$R_{time,check}(n,i) = \frac{i(9n+6)\ T_{check}}{T_{loop}\ n + T_{other} + 9T_{check}\ n\ i + 9T_{data}\ n + 6T_{check}\ i + 6T_{data}} \tag{5.25}$$

Figure 5.20 shows the execution time ratios according to these formula. The left figure shows them depending on the number of active weaving instructions i and constant skeleton loop iterations of $n = L = 1$. The right figure shows the respective execution time ratios for variable number of loop iterations n and a constant number of weaving instructions $i = I = 10$. This graphic shows all relative contributions to execution time together, which means that they always sum up to 100

The graphs based on theoretical considerations shown in Figure 5.20 resemble Figure 5.19 very well. It is directly generated from the measured data. This demonstrates

that the theoretical model for execution time characteristics of the AOP enhanced composition engine implementation very well reflects reality as measured. Furthermore, the derived theoretical model is capable of distinguishing the unique execution time contributions from several major parts of the composition and AOP execution process. This proves to be highly useful, because it allows to predict the performance of the execution engine for a broader range of composite applications. It furthermore allows to identify and discuss general characteristics of the AOP solution.

The examples used in the measurements were deliberately created with focus on composition and weaving execution in order to minimize external disturbances. For this reason, also the base composite application is kept minimalistic. It does not execute much service logic and is therefore very fast. Executing constituent services usually needs much more time than composing them. The used constituent service executes much faster than usual services, because it does not perform any action other than being invoked and exit immediately. Furthermore, it is an internal service. This means, it is invoked entirely through internal calls within the Java execution environment, without involving time consuming external protocol stacks.

Another external activity of the composition engine is the query sent to the service database as part of service selection. The composition engine caches these database queries and the resulting list of service candidates. As the measurements always request the same constituent service from a particular selection constraint, only for the first out of many thousand measurements an external query to the service database is actually sent.

In the measured examples the time for composition and aspect weaving appears to have a high share in the overall execution of the composite application. Using real constituent services and in particular those based on external protocols, such as SOAP or SIP would considerably increase the overall execution time T_{base}, while execution time for aspect weaving still stays as measured. When using an external web-service the execution of a constituent service can realistically add several 10 ms for each invocation. If the base constituent service takes 10 ms for execution and with n=10 loops in the example the total execution time of the composite application would therefore be more than 100 ms instead of 0.21 ms and become the dominant factor. In this example, the pure composition effort becomes a neglectable contributer to execution time and aspect weaving would still contribute 8.5 ms. This means, that using AOP in a real service scenario would only have increased the application execution time by less than 8.5%.

This discussion shows that the final decision if the implemented AOP framework is acceptable or not depends to a great extent on the composite applications and the constituent services being used. Using AOP will increase the absolute latency from the execution engine considerably compared to the basic composition. The relative increase of application execution performance might on the other hand be not too big and acceptable.

5.2.4 Latency of Advice Execution

In Chapter 5.2.3 the performance of the weaving logic was investigated with respect to processing join-points and checking weaving conditions. No advice was executed in those

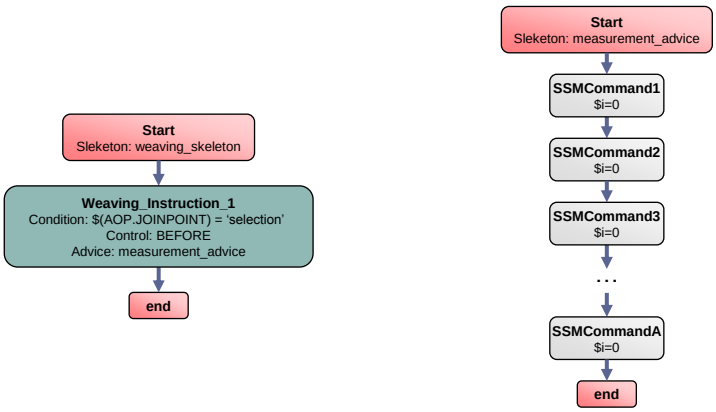

Figure 5.21: Weaving and advice skeletons for the measurement of advice execution

measurements. The performance contribution of invoking and executing advice being invoked and executed will be investigated in this chapter.

The measurements start using the same base composite service as in previous measurements. It contains a single service template in a loop. The weaving instruction is however different now. It checks for the join-point 'service selection' with advice execution 'before' the join-point. In the base skeleton this join-point exists once within the loop.

Advice is implemented by means of a composition skeleton. It contains a variable number A of SSM command elements. For $A = 0$ the advice is empty in the sense that the advice skeleton only consists of a start and an end element. This advice would then be started without executing anything else. For $A > 0$ the advice contains SSM commands assigning a value to a variable.

A variation of SSM commands in the advice is done in order to variate the number of additional join-points contributed to the execution by the advice. This means that also within an advice weaving is applied and all weaving conditions are checked. The used weaving instruction and advice are shown in Figure 5.21. Due to the used weaving instruction the advice will be executed once per base skeleton loop iteration. For this measurement there is also only one weaving instruction loaded, thus, $I = 1$. The weaving conditions are written in a way that no weaving check in the advice execution is matching.

This measurement will determine T_{advice}, which is the effort the advice execution has within the composite application. The total execution time is:

$$T_{total} = T_{base} + T_{weaving} + T_{advice} \tag{5.26}$$

Where the time needed for weaving $T_{weaving}$ is a function of the number of loaded weaving instructions and it consists of the time $T_{preparation}$ and $T_{conditions}$. $T_{preparation}$ expressed the effort spent on preparing all data needed for weaving condition checks. It is a function of the number of executed weaving-points and constant for a given skeleton. $T_{conditions}$ is the effort spent on checking weaving conditions. It is a function of the number of loaded weaving instructions for any given skeleton.

The time T_{advice} expresses the overall execution time spent on advice execution. The same AOP enabled composition engine executes also the advice skeleton. It is therefore again subject to weaving. This means:

$$T_{advice} = T_{adviceweaving} + T_{advicebase} = T_{advicepreparation} + T_{adviceconditions} + T_{advicebase}$$

The times $T_{advicepreparation}$ and $T_{adviceconditions}$ are the corresponding times to $T_{preparation}$ and $T_{conditions}$ but for those preparations and condition checks that happen while executing an advice. Respectively the time $T_{advicebase}$ corresponds to T_{base}. This means for the total execution time:

$$T_{total} = T_{base} + T_{preparation} + T_{conditions} + T_{advicepreparation} + T_{adviceconditions} + T_{advicebase} \tag{5.27}$$

The numbers of weaving points executed in the base skeleton is W_{base} and the number executed in the advice is W_{advice}. Respectively the number of condition checks in the base skeleton execution is C_{base} and in the advice execution is C_{advice}:

$$W_{total} = W_{base} + W_{advice}$$
$$C_{total} = C_{base} + C_{advice}$$

Advice execution is not different from any other skeleton execution. This implies, that the time needed at each weaving-point for preparing the AOP specific shared state T_{data} and the time for each single weaving condition check T_{check} is the same as in the base skeleton, thus:

$$T_{total} = T_{base} + W_{base}T_{data} + C_{base}T_{check} + W_{advice}T_{data} + C_{advice}T_{check} + T_{advicebase}$$

The values for T_{data} and T_{check} were calculated in Chapter 5.2.3 and values for T_{base} were measured in Chapter 5.2.2. All these values from the previous chapters can be re-used here, because the same composite service is used.

The advice contains a variable number A of SSM commands. With $A = 0$ there are only the start and end elements. This most simple skeleton contains two weaving points. With each SSM command additional 2 weaving points are added. This means the number of weaving points in the advice is $2A + 2$. This expresses the number of weaving points executed while one single advice execution. The advice is executed once per base skeleton loop. Therefore, the total number of executed advice weaving-points is $W_{advice} = L(2A+2)$, with L being the number of loop iterations. For only one weaving instruction there is only one condition check per weaving point $C_{advice} = W_{advice}$.

In this measurement the number of base skeleton iterations is varied between $L = 0$ and $L = 10$. The result of the measurement is shown in Figure 5.22. For $L = 0$ the

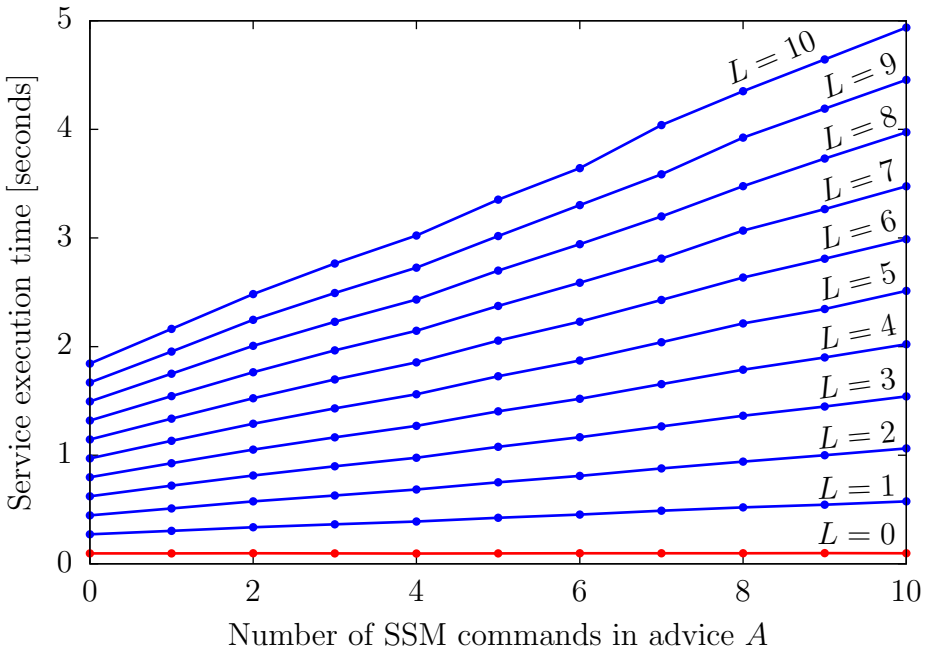

Figure 5.22: Execution time dependent on advice complexity

base skeleton loop is not executed, and therefore, also the nested service template is never executed. Consequently, this means that for $L = 0$ there is no matching weaving condition and no advice is started. Consequently, the advice complexity as expressed by the number A of SSM command elements does not matter. Measurements show a constant execution time. When advice is executed for $L > 0$ the execution time depends linearly on the advice complexity.

$$T_{advicebase,single} = T_{ssm}\ a + T_{start}$$

Where T_{ssm} is the base execution time of each SSM command and T_{start} is the time needed for starting the skeleton execution, which corresponds to executing the start and end elements. Based on the measurements their values can be determined as $T_{ssm} = 0.0193\,\text{ms}$ and $T_{start} = 0.5745\,\text{ms}$.

The overall contribution from basic execution of advice skeletons is then dependent on the number of advice invocations a:

$$T_{advicebase}(n, a) = n(T_{ssm}\ a + T_{start})$$

The share of partial execution time R_{time} was introduced in Chapter 5.2.3 by equation (5.21). With advice being executed the execution time contributions from advice skeleton execution and advice weaving can be considered. The total time T_{total} is according to (5.27) and for the skeletons and weaving instructions used here the following formula can be found:

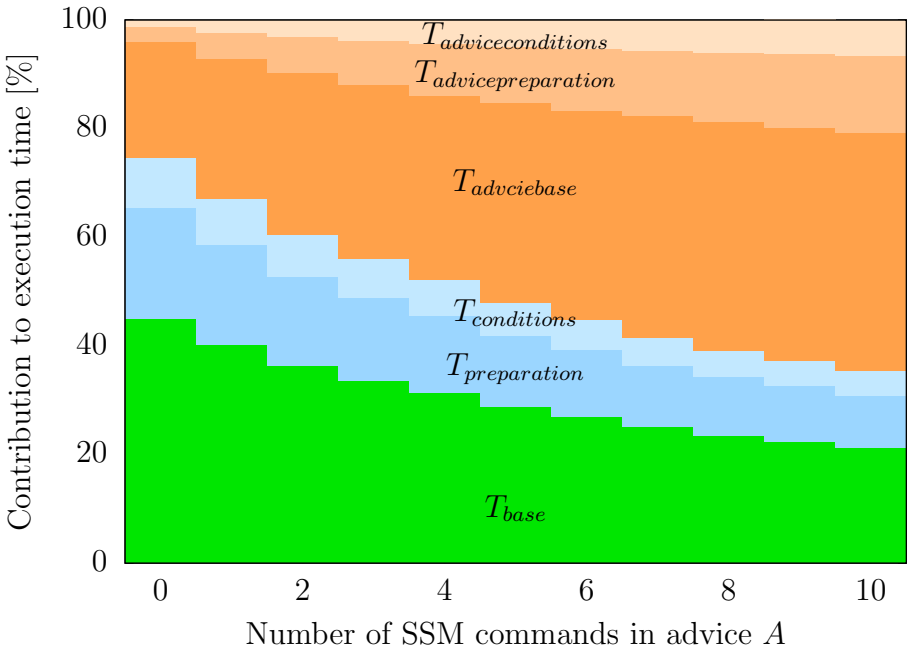

Figure 5.23: Execution time contributions depending on advice complexity (measurement)

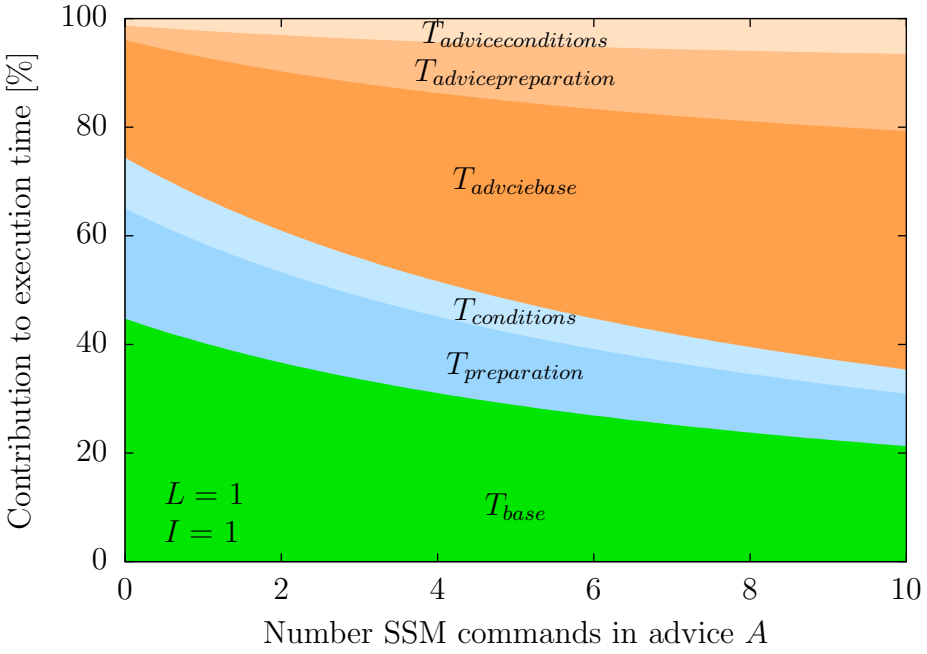

Figure 5.24: Execution time contributions depending on advice complexity (model)

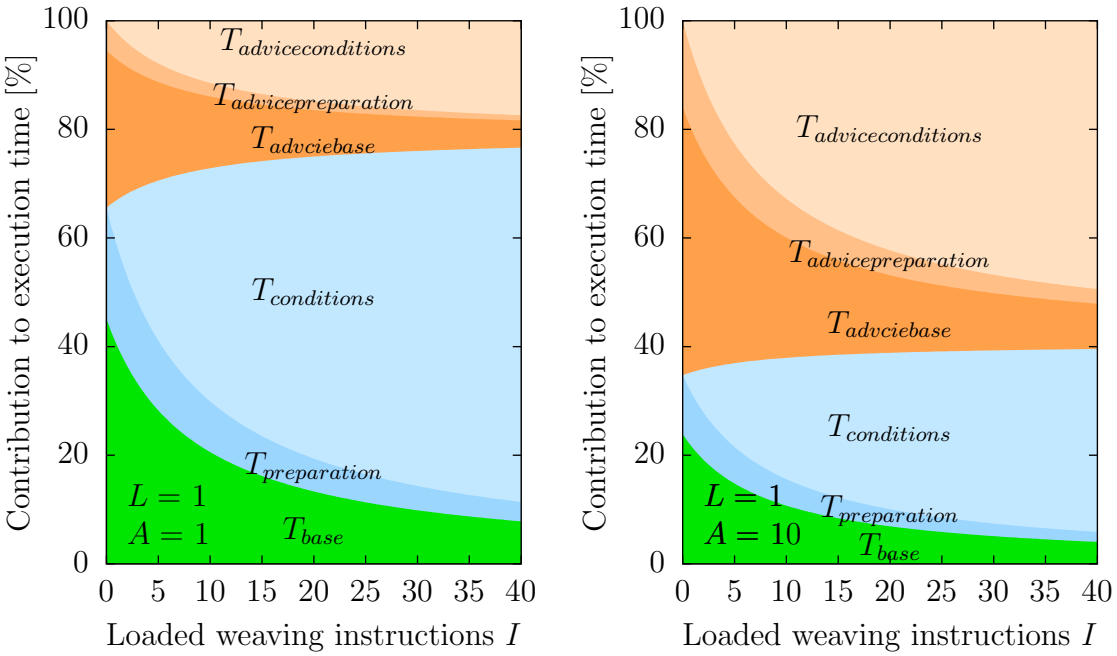

Figure 5.25: Execution time contributions depending on loaded weaving instructions

$$T_{advicepreparation}(n, a) = n(2a + 2)T_{data}$$
$$T_{adviceconditions}(n, i, a) = ni(2a + 2)T_{check}$$
$$T_{advicebase}(n, a) = n(T_{ssm}\ a + T_{start})$$

This allows to generate the diagrams shown in Figure 5.24 and Figure 5.25. They show the relative contributions to overall execution time for the skeletons and weaving instructions as used in the measurements. The diagram in Figure 5.24 is the equivalent to the diagram shown in Figure 5.23 that was created directly from the measurements. It shows the increasing contribution of more complex advice skeletons. Please note that not only the execution of the advice elements as such contributes to the execution, but also the weaving applied to the advice.

The diagrams in Figure 5.25 show the influence of the number of loaded weaving instructions on the execution time. The upper diagram is for a relatively simple advice with $A = 1$ and the bottom diagram shows the difference for a more complex advice with $A = 10$. In both cases the effort for checking the weaving instructions is becomes the main contributor to the execution time. This confirms again earlier findings. Here, especially the weaving within the advice has a major share of the execution effort. Table 5.4 summarizes key results of this measurement.

Parameters	$A = 1$, $L = 1$		$A = 1$, $L = 10$	
Base skeleton execution: T_{base}	0.122 ms	40.3 %	0.710 ms	32.8 %
Weaving in base skeleton:				
Join-Points J_{base}	7		34	
Weaving-Points W_{base}	15		96	
Weaving Checks C_{base}	15		96	
$T_{preparation} = W_{total}\ T_{data}$	0.056 ms	18.3 %	0.355 ms	16.4 %
$T_{conditions} = C_{total}\ T_{check}$	0.026 ms	8.4 %	8.161 ms	7.5 %
Advice execution:				
Join-Points J_{advice}	4		40	
Weaving-Points W_{advice}	4		40	
Weaving Checks C_{advice}	4		40	
$T_{advicebase}$	0.078 ms	25.9 %	0.718 ms	33.2 %
$T_{advicepreparation} = W_{advice}\ T_{advice}$	0.015 ms	4.9 %	0.148 ms	6.8 %
$T_{adviceconditions} = C_{advice}\ T_{advice}$	0.007 ms	2.2 %	0.068 ms	3.1 %
T_{advice}	0.100 ms	33.0 %	0.934 ms	43.2 %
T_{total}	0.303 ms		2.163 ms	

Parameters	$A = 10$, $L = 1$		$A = 10$, $L = 10$	
Base skeleton execution: T_{base}	0.122 ms	21.3 %	0.710 ms	14.4 %
Weaving in base skeleton:				
Join-Points J_{base}	7		34	
Weaving-Points W_{base}	15		96	
Weaving Checks C_{base}	15		96	
$T_{preparation} = W_{total}\ T_{data}$	0.056 ms	9.7 %	0.355 ms	7.2 %
$T_{conditions} = C_{total}\ T_{check}$	0.026 ms	4.4 %	8.161 ms	3.3 %
Advice execution:				
Join-Points J_{advice}	22		220	
Weaving-Points W_{advice}	22		220	
Weaving Checks C_{advice}	22		220	
$T_{advicebase}$	0.252 ms	43.9 %	2.519 ms	51.0 %
$T_{advicepreparation} = W_{advice}\ T_{advice}$	0.081 ms	14.2 %	0.815 ms	16.5 %
$T_{adviceconditions} = C_{advice}\ T_{advice}$	0.037 ms	6.5 %	0.374 ms	7.6 %
T_{advice}	0.371 ms	64.6 %	3.708 ms	75.1 %
T_{total}	0.575 ms		4.937 ms	

Table 5.4: Measurement results and contributions to the overall execution time

5.2.5 Differences of Join-Point Types

The measurements so far have focused on the weaving point before the service selection join-point and before an SSM command. The assumption was that all join-points are similar with respect to needed time for executing the weaving and advice. The measurement presented in this chapter verifies this assumption by determining if there is a difference between join-points. For this purpose, the same advice is weaved at different join-points. The used base skeleton, weaving skeleton and advice skeleton is shown in Figure 5.26.

The measured execution times per used weaving-point are shown in Figure 5.27. It shows similar values for different weaving-points. The conclusion is that the performance of weaving and advice execution does not depend on the join-point and also not on the used weaving point. The only significant exception in this measurement is the around weaving at the service selection join-point. It disables the execution of service selection. This will lead to no constituent service being selected and executed. This missing execution is visible in the measurement in a smaller execution time.

The findings of this chapter are valuable for aspect design: It is not necessary to prefer certain weaving-points for performance reasons. In the scope of the verification measurements in previous chapters it means that it is not necessary to repeat the measurements at different join-point types. The results will be the same.

5.2.6 Weaving Ratio

The previous measurements have shown that weaving and in particular the check of weaving conditions is prime contributer to the execution time. With globally valid weaving instructions all conditions are always checked at each weaving point. Consequently, most checks will not match at a particular weaving-point. In order to evaluate the influence on the performance, a measure for successful checks is introduced: The weaving ratio $R_{weaving}$ is the relation between the number of all executed weaving instruction checks C_{all} and those that lead to starting an advice C_{advice}. It is therefore also the ratio between locally

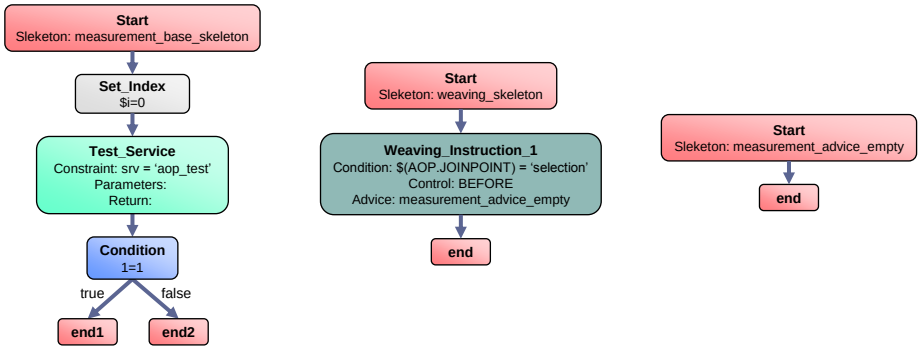

Figure 5.26: Base skeleton, weaving skeleton and advice skeleton for the measurement of weaving at different join-points

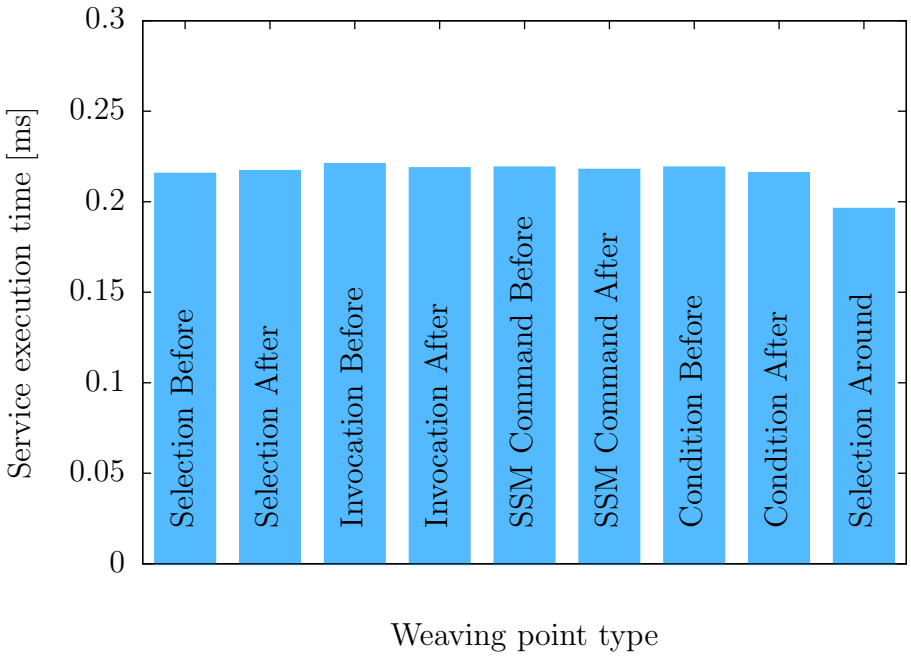

Figure 5.27: Execution time per weaving point type

useless weaving checks and those which lead to execution of wanted logic. This directly expresses a measure of efficiency in the AOP usage.

$$R_{weaving} = \frac{C_{match}}{C_{total}} \tag{5.28}$$

The measurement in Chapter 5.2.4 for $L = 1$ has $C_{base} = 15$ and $C_{advice} = 4$ weaving checks according to Table 5.4. This means a total of $C_{total} = 19$ weaving checks. The advice was only invoked once so that $C_{match} = 1$. The resulting weaving ratio is therefore:

$$R_{weaving} = \frac{C_{match}}{C_{total}} = \frac{1}{19} = 0.0263$$

And for $L = 10$ the total number of matching weaving conditions is $C_{match} = 10$ while the total number of weaving checks is $C_{total} = C_{base} + C_{advice} = 96 + 40 = 136$ the weaving ratio is:

$$R_{weaving} = \frac{C_{match}}{C_{total}} = \frac{10}{136} = 0.0735$$

Table 5.5 shows the weaving ratios for some combinations of advice complexity and base skeleton loop iterations.

Parameters	$A = 1$ $L = 1$	$A = 10$ $L = 1$	$A = 1$ $L = 10$	$A = 10$ $L = 10$
Weaving in base skeleton:				
Join-Points J_{base}	7	7	34	34
Weaving-Points W_{base}	15	15	96	96
Weaving Checks C_{base}	15	15	96	96
Weaving in advice execution:				
Join-Points J_{advice}	4	22	40	220
Weaving-Points W_{advice}	4	22	40	220
Weaving Checks C_{advice}	4	22	40	220
Weaving Check Matches C_{match}	1	1	10	10
Total Weaving Checks C_{advice}	19	37	136	316
Weaving Ratio $R_{weaving}$	0.0526	0.0270	0.0735	0.0316

Table 5.5: Weaving ratio

Within the overall execution of composite applications and aspects, there are useful and not useful parts. Executing the base skeleton is obviously useful, because it is the core business logic of the application. Also executing the advice is useful, because it adds to the business logic. Everything in weaving that does not lead to advice execution is consequently not usefully spent effort. Reducing this not useful part, and thus, bringing the weaving ration up is essential for a more efficient AOP usage.

One observation is that the weaving within an advice may contribute considerably to the total number of weaving checks. Weaving within an advice is rarely useful. Thus, this extra weaving is in most cases unnecessarily spent processing capacity. If, for example, there is no weaving in advice, the weaving ratio in the case of $L = 10$ and $A = 10$ would be 0.104 instead of 0.032.

The weaving ratio directly specifies how often an advice is started. For the example measurements it allows to express the total execution time:

$$T_{total} = T_{base} + R_{weaving}T_{adv} + (W_{base} + R_{weaving}W_{adv})T_{data} + I(W_{base} + R_{weaving}W_{adv})T_{check}$$

Where T_{adv} is the time needed for basic execution of an advice skeleton.

Useful execution time is:

$$T_{total} = T_{base} + R_{weaving}T_{adv} + R_{weaving}W_{base}T_{data} + IR_{weaving}W_{base}T_{check}$$

Not useful execution time is:

$$\begin{aligned} T_{total} = {} & (1 - R_{weaving})W_{base}T_{data} + R_{weaving}W_{adv}T_{data} + \dots \\ & \cdots + I(1 - R_{weaving})W_{base}T_{check} + IR_{weaving}W_{adv}T_{check} \end{aligned}$$

Here all weaving checks within an advice execution session are already considered to be not particularly useful.

5.3 Performance Optimizations

The measurements alongside theoretical considerations in Chapter 5.2 have shown that the performance of composite service execution is considerably decreasing once AOP is used. The execution of aspect weaving rather than basic composition becomes the dominant contributor to execution time. Major contributions to this result are:

- event management infrastructure that reports the execution of a weaving-point as shown in Chapter 5.2.2,
- data handling and weaving condition checks as shown in Chapter 5.2.3
- advice execution including weaving also in advice sessions as shown in Chapter 5.2.4

The major contribution is coming from repeated checking of all weaving conditions at each join-point. Reducing the number of unnecessary weaving condition checks will therefore have a highly positive impact on composition execution time. The weaving ratio introduced in Chapter 5.2.6 is a measure of the efficiency of the aspect implementation.

This chapter proposes and discusses potential variations in the AOP solution in order to improve performance. The theoretical model of Chapter 5.2 did provide a quantified breakdown of the performance cost for a certain action of the integrated composition and weaving execution engine. Based on this model the expected performance gain of each proposed variation in the AOP concept and a respective implementation can be predicted. All proposals are about variations in the AOP concept and not about better implementation techniques or server optimization. There might be improvements possible from that angle, but this is out of scope of this study.

5.3.1 The Original Implementation Without Optimizations

Before optimizations will be proposed and assessed, this chapter introduces the composite application service skeleton, the weaving instructions and the advice skeleton that is used in all discussions. They are shown in Figure 5.28. The base application skeleton consists of tree service templates and three SSM commands and the advice contains three SSM commands. The advice is weaved into the composite application before each service template.

The execution time of the composite application is first measured without weaving instructions loaded. This provides the execution time of just the base application $T_{base} = 0.166\,\text{ms}$. With the weaving instruction loaded, and therefore, with advice being executed, the total execution time is $T_{total} = 0.725\,\text{ms}$.

The components of the total execution time are the base execution times of the composite application T_{base} and the advice $T_{advicebase}$, the time needed for weaving data preparation in the base skeleton $T_{preparation}$ and the advice $T_{advicepreparation}$ and the time needed for executing all weaving checks in the base skeleton $T_{conditions}$ and all weaving checks in the advice $T_{adviceconditions}$.

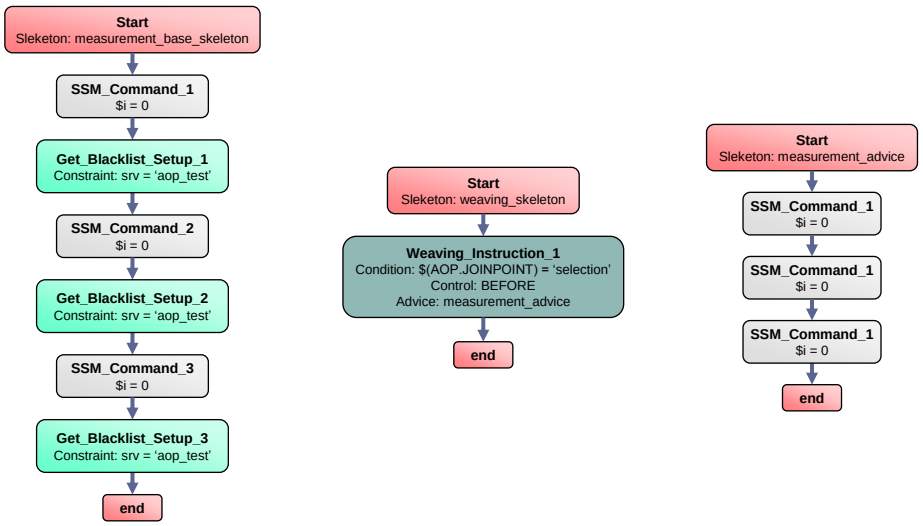

Figure 5.28: Base skeleton, weaving skeleton and advice skeleton for the measurement of weaving at different join-points

$$T_{total} = T_{base} + T_{preparation} + T_{conditions} + T_{advicebase} + T_{advicepreparation} + T_{adviceconditions} \tag{5.29}$$

The used base skeleton contains $J_{base} = 11$ join-points resulting in $W_{base} = 23$ weaving-points. With only one weaving instruction loaded, the number of weaving checks being executed is $C_{base} = IW_{base} = 23$. Here, $I = 1$ is the number of loaded weaving instruction. This allows to calculate the contributions of data preparation and weaving checks in the base skeleton:

$$\begin{aligned} T_{preparation} &= W_{base}\, T_{data} && = 0.0852\,\text{ms} \\ T_{conditions} &= IW_{base}\, T_{check} && = 0.0391\,\text{ms} \end{aligned}$$

The values for T_{data} and T_{check} were determined by measurement in Chapter 5.2.3. They can be re-used here as the same implementation of the respective detailed features is assumed.

The advice skeleton contains 5 join-points with 8 weaving points. The advice is executed once at each service template, and therefore $E = 3$ times. This means in total there are $J_{advice} = 15$ join-points and $W_{advice} = 24$ weaving-points executed. The execution time contributions from data preparation and weaving condition checks in the advice is then:

$$\begin{aligned} T_{advicepreparation} &= W_{base}\, T_{data} && = 0.0889\,\text{ms} \\ T_{adviceconditions} &= IW_{base}\, T_{check} && = 0.0408\,\text{ms} \end{aligned}$$

Parameters		Values	Values
Loaded weaving instructions	I	1	20
Weaving matches	E	3	3
Weaving in base skeleton:			
Join-Points	J_{base}	11	11
Weaving-Points	W_{base}	23	23
Weaving checks	C_{base}	23	460
Weaving in advice execution:			
Join-Points	J_{advice}	15	15
Weaving-Points	W_{advice}	24	24
Weaving checks	C_{advice}	24	480
Total weaving checks	C_{advice}	47	940
Weaving ratio	$R_{weaving}$	0.06382	0.00016
Execution time contributions:			
Base skeleton	T_{base}	0.166 ms	0.166 ms
Weaving data preparation	$T_{preparation}$	0.085 ms	0.085 ms
Weaving condition checks	$T_{conditions}$	0.039 ms	0.782 ms
Advice skeleton execution	$T_{advicebase}$	0.306 ms	0.306 ms
Advice weaving data	$T_{advicepreparation}$	0.089 ms	0.089 ms
Advice condition checks	$T_{advicepreparation}$	0.041 ms	0.816 ms
Total execution time	T_{total}	0.725 ms	2.243 ms

Table 5.6: Parameters and time of composite service execution without optimizations for $I = 1$ and $I = 20$ loaded weaving instructions

Using (5.29) these values allow to calculate the time needed for advice execution $T_{advicebase} = 0.3058\,\text{ms}$. Advice is executed $E = 3$ times. Therefore, each advice skeleton execution contributes approximately $0.1019\,\text{ms}$.

The weaving ratio for this example is calculated following its definition in (5.28):

$$R_{weaving} = \frac{C_{match}}{C_{total}} = \frac{E}{C_{base} + C_{advice}} = 0.06383$$

All values are summarized in Table 5.6. Figure 5.29 shows the projected execution time. It shows the contributions from skeleton and advice execution, weaving data preparation and weaving condition checks. The left diagram shows the relative contributions and the right diagram the corresponding absolute execution time.

Useful business logic is executed in the base skeleton and the advice. The rest of the time is spent on getting the advice executed. This time is what the following optimization proposals try to minimize.

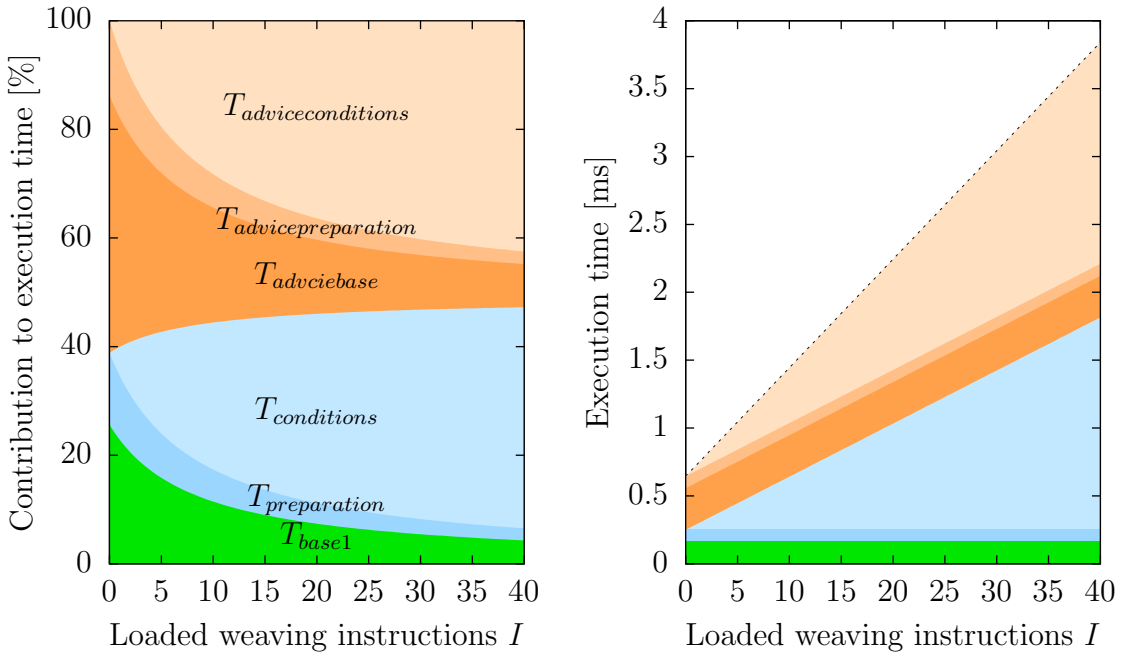

Figure 5.29: Relative and absolute execution time contributions depending on loaded weaving instructions, no optimizations

5.3.2 Disable Weaving in Advice Execution Sessions

Weaving within advice execution sessions means aspects being applied to aspects. This behavior is a direct consequence of advice being implemented based on the same composition language and executed by the same execution engine as base application skeletons. The composition execution engine notifies the weaving engine at all weaving points regardless of their origin. It intentionally does not distinguish between base sessions and advice sessions.

Weaving within advice is a feature that in practice should not be used frequently. Aspects are usually designed against the base application's code. A weaver, that applies it also to other advice is likely to cause complex aspect interaction problems. Therefore, this property most likely causes more harm than it helps. On top of the problems, each weaving-point being checked within advice execution sessions adds to the overall execution time. Therefore, the proposal is to disable weaving in advice execution completely.

The capability of weaving in advice sessions is not really needed for practical use cases. The degree of cross-cutting within advice is likely to be small, because aspects are by definition made to add single concerns each. Under these circumstances, advice code is not a good target for using AOP in order to further reduce cross-cutting. The only case where this appears to be useful would be using an aspect for targeting global changes across many applications including their more specific local aspects.

Another problem is that weaving within advice might make the writing of weaving

Parameters		No optimizations	No weaving in advice sessions
Loaded weaving instructions	I	20	20
Weaving matches	E	3	3
Weaving in base skeleton:			
Join-Points	J_{base}	11	11
Weaving-Points	W_{base}	23	23
Weaving checks	C_{base}	460	460
Weaving in advice execution:			
Join-Points	J_{advice}	15	0
Weaving-Points	W_{advice}	24	0
Weaving checks	C_{advice}	480	0
Total weaving checks	C_{advice}	940	460
Weaving ratio	$R_{weaving}$	0.00016	0.00033
Execution time contributions:			
Base skeleton	T_{base}	0.166 ms	0.166 ms
Weaving data preparation	$T_{preparation}$	0.085 ms	0.085 ms
Weaving condition checks	$T_{conditions}$	0.782 ms	0.782 ms
Advice skeleton execution	$T_{advicebase}$	0.306 ms	0.306 ms
Advice weaving data	$T_{advicepreparation}$	0.089 ms	0 ms
Advice condition checks	$T_{advicepreparation}$	0.816 ms	0 ms
Total execution time	T_{total}	2.243 ms	1.339 ms
Reached reduction			40 %

Table 5.7: Parameters and time of composite service execution without weaving in advice sessions for $I = 20$ loaded weaving instructions

instructions and advice harder. If, for example, the advice uses the same constituent services as the base applications, weaving instructions might match again and weaving goes into a loop. Loop prevention strategies can detect and stop this behavior, but from performance perspective all this would be costly.

An efficient way for inhibiting weaving in advice execution sessions would be by explicitly marking these sessions. The execution and weaving engine can, for example, write a special flag into the shared state that is generated at a join point. AOP specific data is anyway written into these shared state sessions exposing additional run time data. When detecting one of the special AOP variables, for example AOP.JOINPOINT, the composition execution engine could inhibit the generation of weaving-point events. Weaving checks would then never be executed for these sessions.

Introducing a separate flag for this purpose would also have an advantage. It can be set and unset dynamically by advice. This is an additional feature that would, for example, allow advice to stops all further application of aspects within a composition session. Using this mechanism also the base skeleton would be able to temporarily forbid all aspects. This can be useful if, for example, a sequence of critical actions needs to be protected from

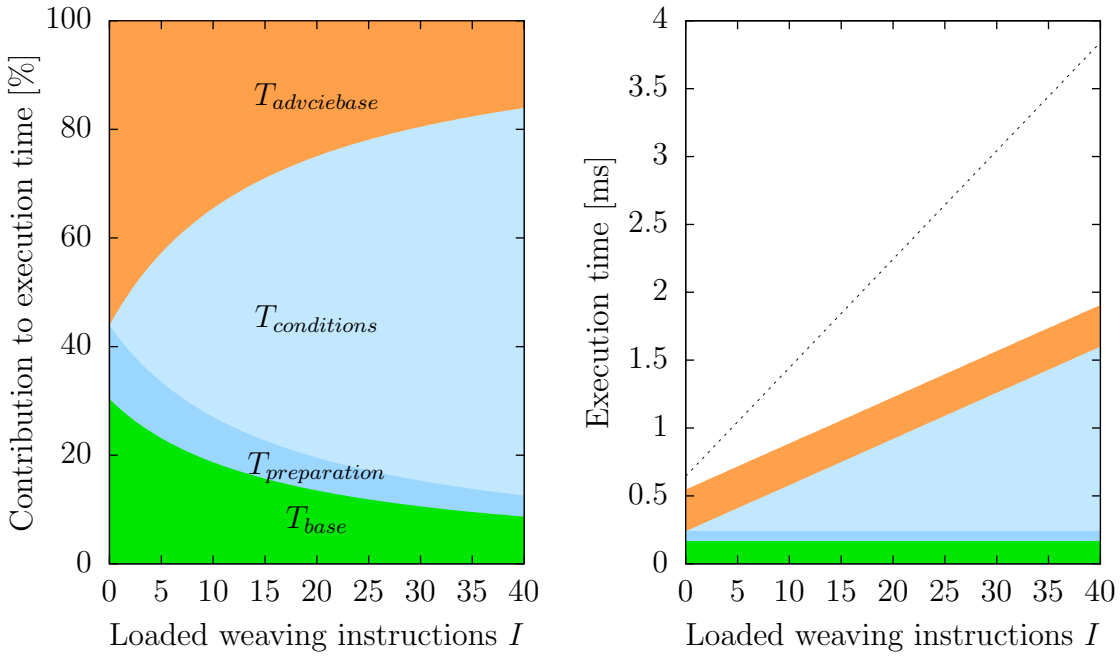

Figure 5.30: Relative and absolute execution time contributions depending on loaded weaving instructions, no weaving in advice execution sesions

aspect weaving.

Without weaving within advice sessions the total execution time can be estimated using (5.29). The components $T_{advicepreparation}$ and $T_{adviceconditions}$ are removed and the total execution time is:

$$T_{total} = T_{base} + T_{preparation} + T_{conditions} + T_{advicebase}$$

The resulting values for the example composite application are shown in Table 5.7 and Figure 5.30. Especially for higher numbers of loaded weaving instructions this proposed change can considerably reduce the overall execution time.

5.3.3 Weaving Instructions Assigned to Composite Applications - Local Weaving Instruction Sets

In the proposed AOP concept, all weaving instructions are globally valid. This means they are applied to any composite application and consequently all weaving conditions are checked at each weaving point. This causes high system load and ultimately a low weaving ratio.

In order to improve this, weaving instructions are divided into sub-sets. Each of these sets is valid for only one composite application or for one execution instance of this application. The weaving engine will only use weaving instructions from the set assigned to the

Parameters		No optimizations	Per session weaving instructions
Assigned weaving instructions	$P_{instructions}$	100 %	40 %
Loaded weaving instructions	I	20	20
Weaving matches	E	3	3
Weaving in base skeleton:			
Join-Points	J_{base}	11	11
Weaving-Points	W_{base}	23	23
Weaving checks	C_{base}	460	184
Weaving in advice execution:			
Join-Points	J_{advice}	15	15
Weaving-Points	W_{advice}	24	24
Weaving checks	C_{advice}	480	192
Total weaving checks	C_{advice}	940	376
Weaving ratio	$R_{weaving}$	0.00016	0.00040
Execution time contributions:			
Base skeleton	T_{base}	0.166 ms	0.166 ms
Weaving data preparation	$T_{preparation}$	0.085 ms	0.085 ms
Weaving condition checks	$T_{conditions}$	0.782 ms	0.313 ms
Advice skeleton execution	$T_{advicebase}$	0.306 ms	0.306 ms
Advice weaving data	$T_{advicepreparation}$	0.089 ms	0.089 ms
Advice condition checks	$T_{advicepreparation}$	0.816 ms	0.326 ms
Total execution time	T_{total}	2.243 ms	1.285 ms
Reached reduction			43 %

Table 5.8: Parameters and time of composite service execution with only a subset of weaving instructions assigned to the application for $I = 20$ loaded weaving instructions

executed application. This limits the number of checks needed at each join-point. Generating and controlling these sets can be an additional feature of the aspect management system.

There are two ways of assigning weaving instruction to execution sessions:

1. Manual: When deploying an aspect, its weaving instructions are uploaded into the execution environment. Part of this process can be the manual assignment of these weaving instructions to composite applications. Especially those aspects that are specifically written for a particular composite application can be exclusively connected to the execution sessions of this application. For others the related weaving instructions are filtered out.

2. Automatic: If an execution session is being created all weaving instructions are analyzed. Only those weaving instructions are loaded into the execution session's local weaving instruction set, that explicitly belong to the application. Also those instructions that do not have a chance to meet their condition are left out. This requires to

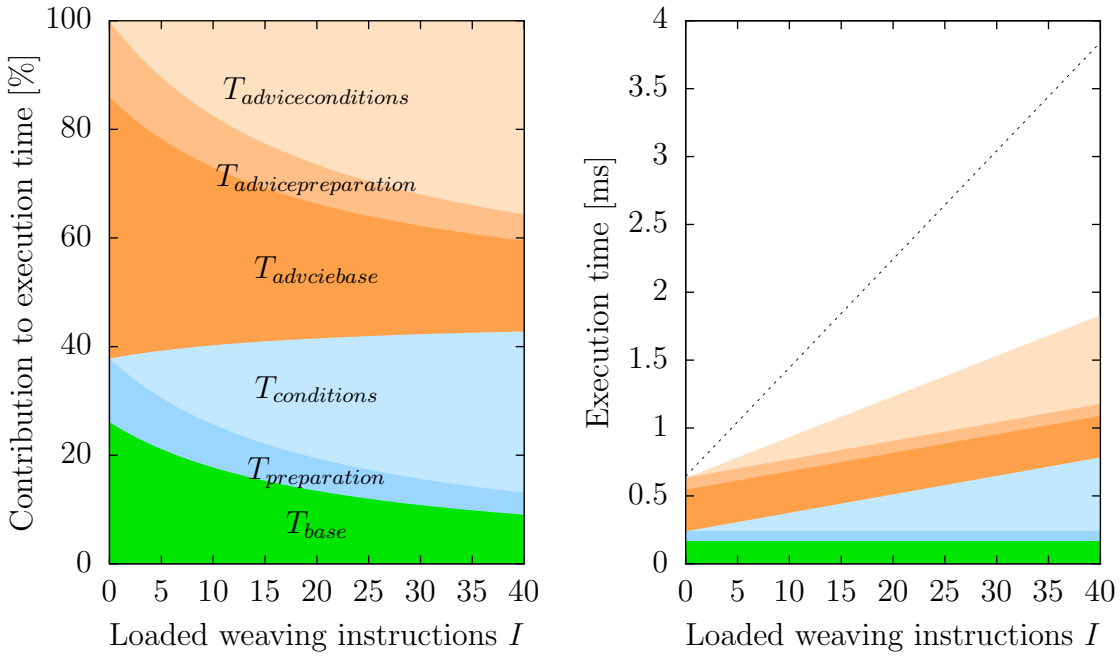

Figure 5.31: Relative and absolute execution time contributions depending on loaded weaving instructions, only a subset of weaving instructions assigned to the application

implement a logic for checking the weaving conditions against the application skeleton in order to find those that can never be true. For example, a condition for a skeleton element that is not present in the skeleton can be left out. With this automatic filtering the full global set of all weaving instructions only need to be consulted once at starting a new execution instance of an application.

3. Both methods can also be combined.

Estimating the consequences for execution time is quite straightforward. The contributions coming from the weaving checks while executing the base skeleton or the advice skeleton will be reduced. The key parameter is the percentage $P_{instructions}$ of weaving instructions that are assigned to the composite application only those are checked. The execution time contribution from weaving condition checks is directly reduced by this percentage factor;

$$T_{total} = T_{base} + T_{preparation} + P_{instructions}T_{conditions} + T_{advicebase} + \ldots$$
$$\cdots + T_{advicepreparation} + P_{instructions}T_{adviceconditions}$$

The estimated results are shown in Table 5.8 and Figure 5.31. In this example $P_{instructions} = 40\%$ is assumed. This means that only 40% of all weaving instructions are applicable for the composite application.

Join-Point	Weaving-Point	Distribution of weaving instructions	
		Equal distribution	Service template centric
Service Selection	Before	0.09 %	57 %
	After	0.09 %	0 %
	Around	0.09 %	6 %
Service Invocation	Before	0.09 %	16 %
	After	0.09 %	4 %
SSM Command	Before	0.09 %	8 %
	After	0.09 %	0 %
Condition	Before	0.09 %	4 %
	After	0.09 %	0 %
Start Element	After	0.09 %	4 %
End Element	Before	0.09 %	1 %

Table 5.9: Relative distribution of weaving instructions

5.3.4 Weaving Instructions per Weaving-Point Type

Using the proposal of Chapter 5.3.3 each composite application has one set of applicable weaving instructions. These weaving instructions can be further analyzed before execution. For example, a part of the weaving condition that is usually present is the join-point type. Furthermore, the weaving instruction determines if the advice shall be applied before, after or instead of the join-point. Both together directly identify the only weaving-points for which this weaving condition can match. Checking this condition at other weaving points is an unnecessary activity.

The proposed enhancement is to split up the overall set of weaving instructions into separate sets per weaving-point type. These more specific sets per weaving point are used instead of a single set per composite application. This creates an additional level of pre-distributing the weaving instructions. The first level is to assign weaving instructions to applications and the second level is to further assign them to separate sets per weaving point. This means at each weaving point only those checks are made that have a chance to be positively evaluated. The idea is that every part of the weaving condition that can be predetermined without the need of run time data is preprocessed in order to keep the effort at execution time low.

This proposal introduces an automatic preprocessing of weaving instructions. By assigning the weaving instruction to a weaving point set, the part of the condition based on the 'aop.joinpoint' variable is already checked. Thus, this partial condition can be removed from the weaving instruction. This reduces also the effort to check the rest of the condition at run time. Depending on the details of the implementation, this can be done already when deploying the aspect. If a weaving-instruction is capturing several weaving-points, it would be duplicated and added to all respective sets. However, in practice this can be expected to be rare case. Usually weaving conditions are highly specific about the type of weaving point they can exclusively be applied to.

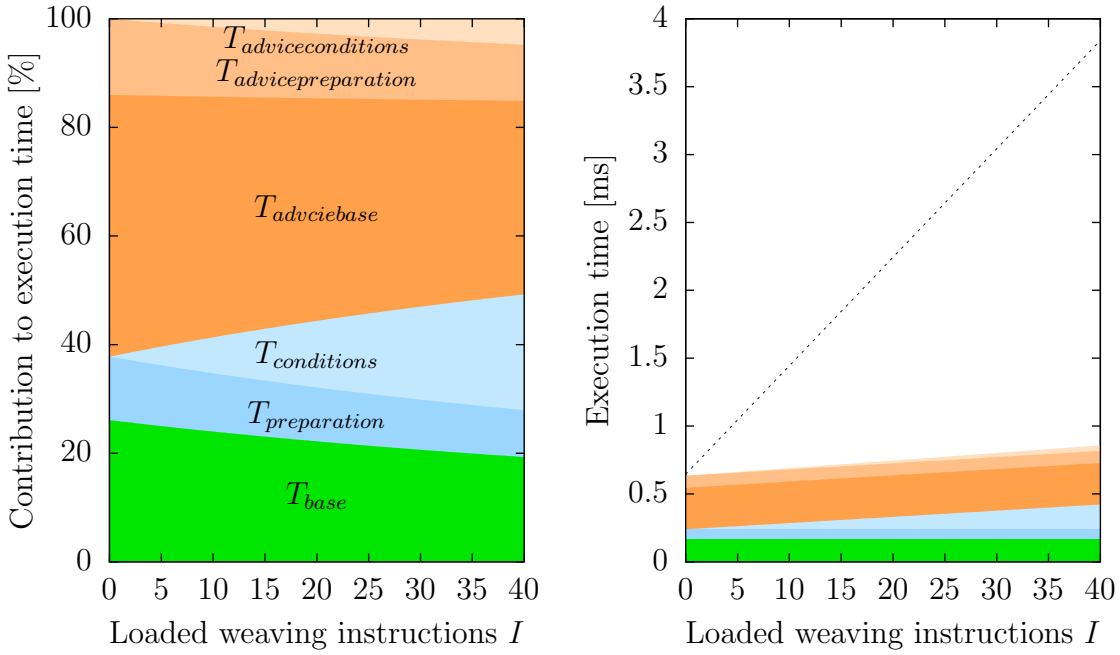

Figure 5.32: Relative and absolute execution time contributions depending on loaded weaving instructions, weaving instruction sets per weaving-point type, equal distribution

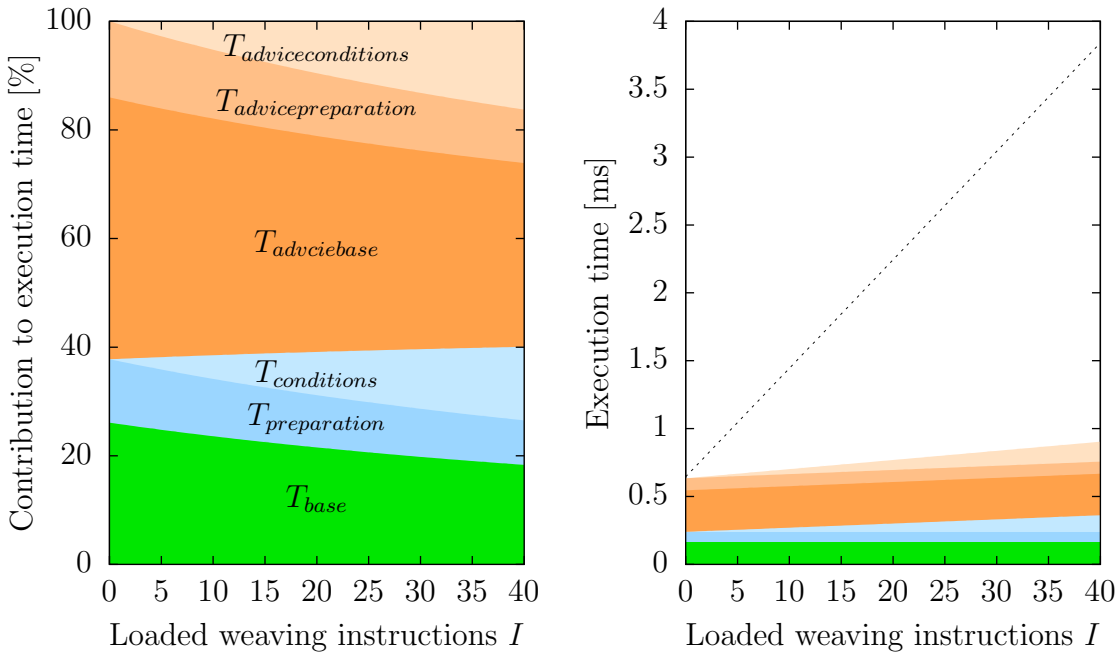

Figure 5.33: Relative and absolute execution time contributions, weaving instruction sets per weaving-point type, service template centric distribution

Parameters		No optimizations	Per weaving-point type weaving instructions	
			Equal distribution	Service template centric
Assigned weaving instructions	$P_{instructions}$	100 %	100 %	100 %
Loaded weaving instructions	I	1	20	20
Weaving matches	E	3	3	3
Weaving in base skeleton:				
Join-Points	J_{base}	11	11	11
Weaving-Points	W_{base}	23	23	23
Weaving checks	C_{base}	460	41.8	55.6
Weaving in advice execution:				
Join-Points	J_{advice}	15	15	15
Weaving-Points	W_{advice}	24	24	24
Weaving checks	C_{advice}	480	130.9	52.2
Total weaving checks	C_{advice}	940	172.7	107.8
Weaving ratio	$R_{weaving}$	0.00016	0.00087	0.00139
Execution time contributions:				
Base skeleton	T_{base}	0.166 ms	0.166 ms	0.166 ms
Weaving data preparation	$T_{preparation}$	0.085 ms	0.085 ms	0.085 ms
Weaving condition checks	$T_{conditions}$	0.782 ms	0.071 ms	0.095 ms
Advice skeleton execution	$T_{advicebase}$	0.306 ms	0.306 ms	0.306 ms
Advice weaving data	$T_{advicepreparation}$	0.089 ms	0.089 ms	0.089 ms
Advice condition checks	$T_{advicepreparation}$	0.816 ms	0.223 ms	0.089 ms
Total execution time	T_{total}	2.243 ms	0.939 ms	0.829 ms
Reached reduction			58 %	63 %

Table 5.10: Parameters and time of composite service execution with weaving instruction sets per join-point type for $I = 20$ loaded weaving instructions

For evaluating the effect of this optimization it is necessary to consider the distribution of weaving instructions per weaving-point. Here two example distributions will be analyzed. The first is an equal distribution. This means that the set for each weaving point contains the same number of weaving instructions.

In practical use cases the aspects can be expected to focus on service templates while other weaving-points are used to much smaller extent or not at all. this is reflected by the second distribution of weaving instructions. Table 5.9 shows the two example distributions that are used here.

Please note that the proposed distribution of weaving instructions in sets per weaving point type can be reached by only analyzing the weaving instructions independent of the deployed composite applications. Therefore, the distribution into these weaving sets can be done at load time of the weaving instructions. This does not break the concept of globally valid weaving instructions.

The expected performance figures for this enhancement are summarized in Table 5.10

Parameters		No optimizations	Diabled weaving-point
Assigned weaving instructions	$P_{instructions}$	100 %	100 %
Loaded weaving instructions	I	20	20
Weaving matches	E	3	3
Weaving in base skeleton:			
Join-Points	J_{base}	11	11
Weaving-Points	W_{base}	23	20
Weaving checks	C_{base}	460	400
Weaving in advice execution:			
Join-Points	J_{advice}	15	15
Weaving-Points	W_{advice}	24	24
Weaving checks	C_{advice}	480	480
Total weaving checks	C_{advice}	940	880
Weaving ratio	$R_{weaving}$	0.00016	0.00017
Execution time contributions:			
Base skeleton	T_{base}	0.166 ms	0.166 ms
Weaving data preparation	$T_{preparation}$	0.085 ms	0.074 ms
Weaving condition checks	$T_{conditions}$	0.782 ms	0.680 ms
Advice skeleton execution	$T_{advicebase}$	0.306 ms	0.306 ms
Advice weaving data	$T_{advicepreparation}$	0.089 ms	0.089 ms
Advice condition checks	$T_{advicepreparation}$	0.816 ms	0.816 ms
Total execution time	T_{total}	2.243 ms	2.130 ms
Reached reduction			5 %

Table 5.11: Parameters and time of composite service execution with disabled weaving point for $I = 20$ loaded weaving instructions

and in Figures 5.32 and 5.33. Even without reducing the total number of weaving instructions and without limiting the usage of aspects, this method will potentially lead to substantial improvements in run time performance.

5.3.5 Disable Weaving-Points

Some weaving points have higher practical value for aspect development than others. For example, the weaving points related to service templates are most likely the target of the majority of weaving conditions. This is a direct consequence of this join-point being directly related to the main purpose of service compositions: select and execute constituent services. Other weaving-points, such as the one associated with the end skeleton element, are seldom used. If they would not exist, the consequences and limitations for aspect design might not be critical. It would therefore make sense to disable these weaving-points. This means disabling the generation of the related weaving-point event.

The following implementation can be used in order to reach the proposed behavior:

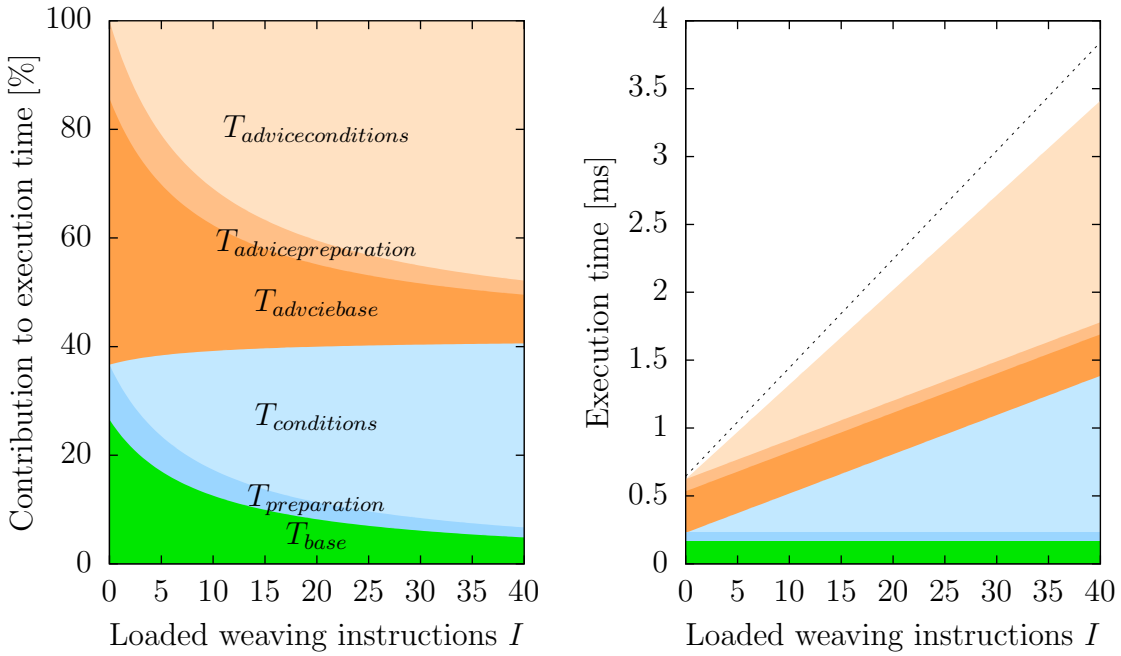

Figure 5.34: Relative and absolute execution time contributions depending on loaded weaving instructions, disable weaving-point

When processing a weaving-point, the composition execution engine checks the related set of weaving instructions. The weaving event is only issued, if the set is not empty. The empty set would anyway lead to no weaving condition checks, but not issuing the event would also save the effort for event handling and for preparing the AOP specific shared state data context.

This implementation utilizes the sets of weaving instructions introduced in Chapter 5.3.4. It dynamically disables individual weaving points depending on need. This means the weaving point is not removed in general from the join-point model. It is still available to the aspect developer but removed if not in use. Alternatively it would be possible to remove unimportant weaving points also from the join-point model.

Table 5.11 and Figure 5.34 show the expected improvement if a weaving point is removed, which is executed 3 times. In the example service in Figure 5.28 this can, for example, be one of the weaving points of the service template.

5.3.6 All Optimizations Combined

In this chapter all proposed performance optimizations are combined. The result is a substantial improvement in execution time as shown in Table 5.12 and Figure 5.35. The same example parameters and assumptions, which were chosen for evaluating individual optimizations, are also applied to the assessment of their combined performance impact. The model calculation shows for $I = 20$ loaded weaving instructions a total execution time

Parameters		No optimizations	Diabled weaving-point
Assigned weaving instructions	$P_{instructions}$	100 %	100 %
Loaded weaving instructions	I	20	20
Weaving matches	E	3	3
Weaving in base skeleton:			
Join-Points	J_{base}	11	11
Weaving-Points	W_{base}	23	20
Weaving checks	C_{base}	460	22.2
Weaving in advice execution:			
Join-Points	J_{advice}	15	0
Weaving-Points	W_{advice}	24	0
Weaving checks	C_{advice}	480	0
Total weaving checks	C_{advice}	940	22.2
Weaving ratio	$R_{weaving}$	0.00016	0.00674
Execution time contributions:			
Base skeleton	T_{base}	0.166 ms	0.166 ms
Weaving data preparation	$T_{preparation}$	0.085 ms	0.074 ms
Weaving condition checks	$T_{conditions}$	0.782 ms	0.038 ms
Advice skeleton execution	$T_{advicebase}$	0.306 ms	0.306 ms
Advice weaving data	$T_{advicepreparation}$	0.089 ms	0 ms
Advice condition checks	$T_{advicepreparation}$	0.816 ms	0 ms
Total execution time	T_{total}	2.243 ms	0.583 ms
Reached reduction			74 %

Table 5.12: Parameters and time of composite service execution with all optimizations combined for $I = 20$ loaded weaving instructions.

of $T_{total} = 0.583\,\text{ms}$. This means an improvement of 73%

No weaving in advice sessions immediately removes much unnecessary processing. In this example about half of the overall weaving checks originate in the advice session, and therefore, they are removed.

The biggest improvement without removing features of the AOP solution is contributed by the management of weaving instruction sets. Weaving instructions are not treated globally applicable any more. Instead, smart offline preprocessing reduces the effort that is needed at run time.

The result of these considerations is a clear recommendation to implement the proposed optimizations.

5.4 Summary and Assessment of Findings

The usefulness of the introduced AOP semantics and syntax could be demonstrated by typical use cases examples. For these use cases the proposed AOP concepts were used to

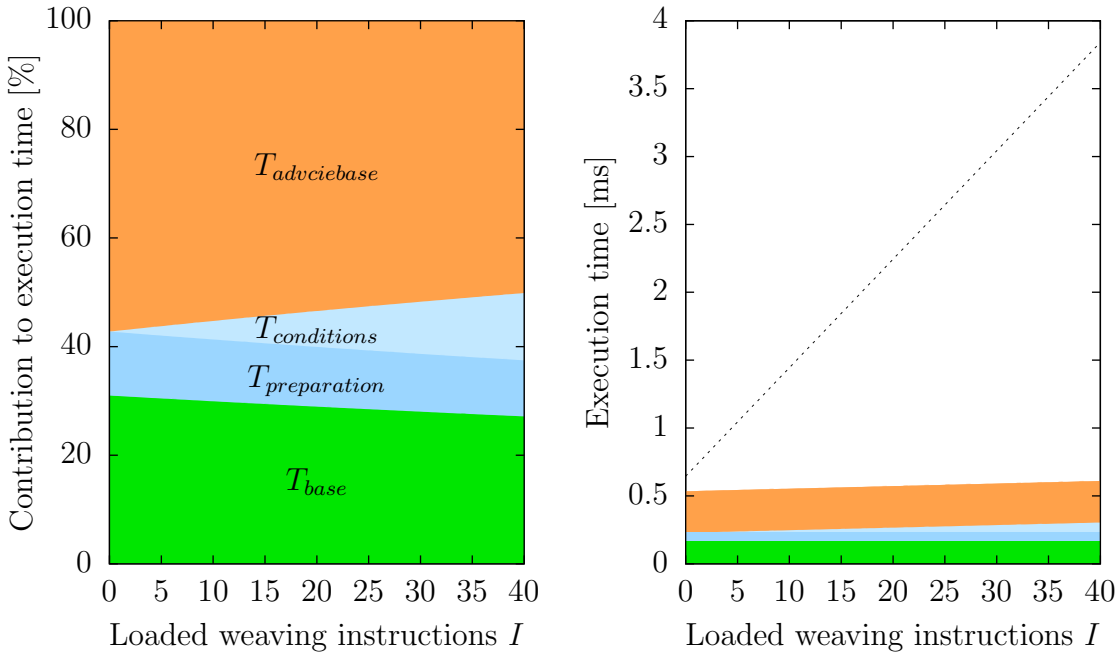

Figure 5.35: Execution time contributions depending on loaded weaving instructions, all optimizations combined.

implement variations in composite applications. This frequently means to interfere with the basic purpose of service composition: to determine which constituent service shall be invoked and when. It could be demonstrated, that the proposed AOP concept provides sufficient and easy to apply tools for changing constituent services including a change of their interface.

The major assessment of the verification deals with the performance of the proposed AOP system. Online waving was expected to come with a considerable performance cost and this prediction could clearly be verified. The composition latency that originates from online weaving can easily exceed the execution of the base composition skeleton. The execution of the wanted additional functionality in the composite application as such by means of advice is not significantly different from being implemented natively in the original skeleton. The latency cost originates in the weaving process and the related creation and management of additional advice execution sessions. Especially checking the weaving instructions at each weaving-point introduces a major impact on execution time.

However, the latency introduced by composition and weaving is still small compared to the latency that is typically needed for executing typical constituent services. The relative latency penalty of composition execution was always much smaller than the constituent service execution. This is still valid using online weaving. The overall application latency still depends mainly on the type and number of constituent services being used.

The absolute latency for executing the composition including aspect weaving is within a few milliseconds. This result could be reached using fast dummy constituent services. The

Optimization	Total execution time T_{total}	Relative improvement	Conceptual limitation
No optimization (main proposed solution)	2.243 ms	-	No
No weaving in advice sessions	1.339 ms	40 %	Yes
Only a subset of weaving instructions assigned to application	1.285 ms	43 %	No
Weaving instructions sets per join-poin type	0.829 ms	63 %	No
Disable weaving points	2.130 ms	5 %	Yes
All optimizations combined	0.583 ms	74 %	Yes

Table 5.13: Overview of improvement proposals and their impact on execution time

absolute latency penalty of AOP stays constant no matter what service technology was used. In practical use cases, when constituent services are invoked by means of external protocols, such as SOAP or SIP, the latency for service invocation can easily be in the range of 100 ms or more per constituent service invocation. This shows that even with the use of AOP, the composition execution time is still at least a magnitude smaller than the expected execution of the constituent services.

Three constituent SOA Web-Service combined easily require combined execution time of 300 ms. The execution of the example skeleton with three service templates without AOP takes 0.166 ms, and thus, composition causes only 0.05% of the overall application execution. Using AOP with the same example composite application required an execution time without considering the constituent services of 2.243 ms. The composition execution with AOP online weaving and without any of the proposed improvements would therefore cause 0.7% of the overall execution time. Although 0.7% is already much bigger than 0.05%, it is still small compared to the overall composite application execution time.

The performance measurements allow to clearly identify what components of weaving execution do contribute and to which extent to the composition execution latency. This did allow to propose variations to the implementation that are expected to improve the execution latency. Based on the previous measurements it was also possible to quantify the impact of each implementation variation. The results show great potential for further improving the performance. Table 5.13 summarizes the findings for the proposed improvements. It also mentions if a proposed optimization would mean reduced capabilities within the Aspect Oriented Programming concept.

The main strategy for improvements is the reduction of online weaving checks. Checks without any major practical value were removed and as many decisions as possible were moved to offline execution. Especially an offline categorization of weaving instructions reduces many unnecessary checks to be executed online. The result is still online weaving, but with offline decision support and a much improved performance without loosing the flexibility and dynamic behavior of online weaving.

Chapter 6

Conclusions & Future Work

Many examples were provided showing that cross-cutting concerns are a frequent and typical challenge for application development in the telecommunication domain. Many of these concerns are directly related to the business environment of the composite application. They deal, for example, with charging and billing, quality of service and SLA management, policy enforcement and fault detection and correction. A technique, such as Aspect Oriented Programming is therefore highly relevant for more efficient development and management of the applications.

The Ericsson Composition Engine provides a service composition environment that, unlike BPMN and BPEL, was specifically designed for composing telecommunication applications. It is able to operate as integral part of the IMS/SIP service chain, but it also allows to work with a broad range of heterogeneous services that are typical for business processes and enterprise applications. The composition execution operates on a high level of abstraction that is to a great extent agnostic of the service technology. Because of these properties it is an excellent base for developing an Aspect Oriented Programming paradigm that particularity targets telecommunication applications.

6.1 Solution Summary

A lean join-point model was introduced. It captures the main semantic elements and composition execution actions in order to provide sufficient anchor points for aspect development. Furthermore, advice is implemented using the same mostly graphical composition language as the base application. The connection of the composition session to weaving and advice execution is reached through join-point events generated in the composition execution. Consequently, the weaving engine acts as event handler.

Weaving instructions are based on extensions of the graphical skeleton language. Pointcuts are therefore loaded by executing a special composite application. This is combined with an aspect management system included into the integrated development environment of the composition engine.

The weaving process and also the executed advice can access the full shared state of

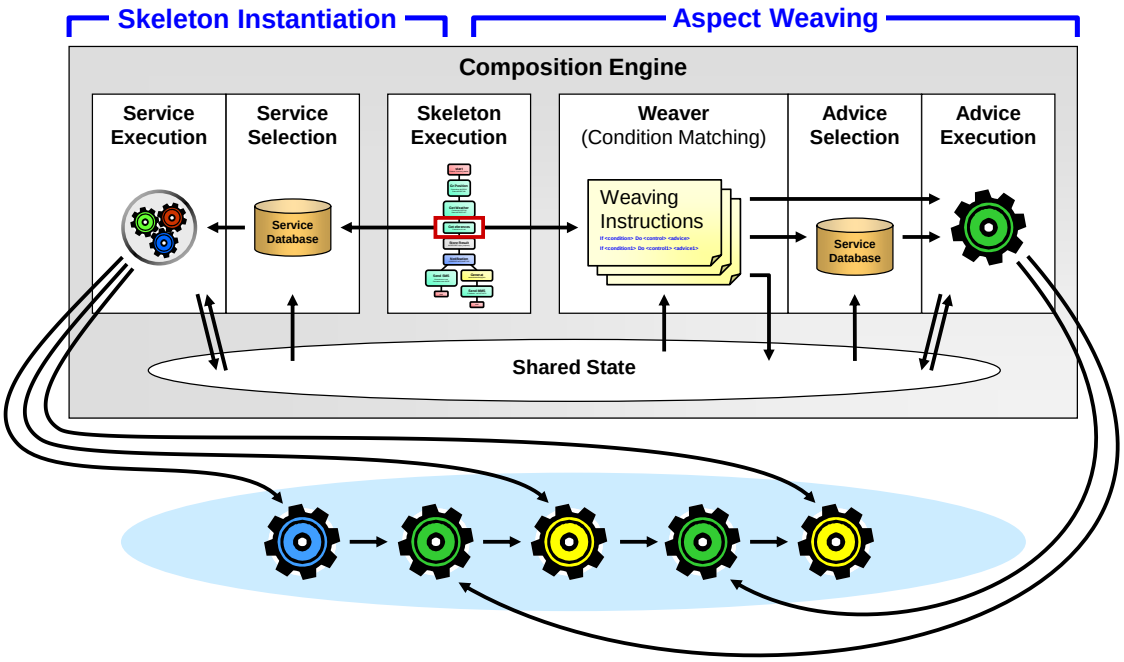

Figure 6.1: AOP enhanced composition engine

a composite application's run time context. Furthermore, selected data from the internal run time of the composition engine is available to advice execution and weaving. Consequently, the decision if advice needs to be started can dynamically be taken at run time. Furthermore, advice being weaved in at a join-point has rich possibilities to modify the composition execution. For example, it can influence the selection of composite services by modifying service selection constraints. It can completely override the default service selection mechanism. This is accompanied by the possibility to change the service API. Constituent services can therefore be completely replaced or even skipped and additional new constituent services might be added.

The developed AOP solution interacts directly with the constraint based composition execution mechanism. Furthermore, advice is expressed as additional composition skeletons. This means that the AOP solution inherits the high abstraction level of the composition mechanism. On that level, the technological details of constituent services are not visible. The resulting ability to natively compose heterogeneous services is important for telecommunication environments and preserved with the AOP solution.

While service selection and run time data can be flexibly modified, the possibility to apply structural changes to the composition is limited. It is possible to override skeleton elements and branching conditions can be changed. On the other hand it is not possible to insert new branches into the base application's skeleton. This is a limitation that has not much relevance for practical use cases. In none of the typical use case examples of Chapter 5.1 did extensive structural changes appear beneficial.

Figure 6.1 shows the combined composition execution and aspect weaving engine according to the proposed solution architecture. Both contribute to the selection and serialization of the constituent services of a composite application. Furthermore, the skeleton of the application is the central control program for both sides. The composition execution evaluates constraints from service templates and ultimately contributes instances of constituent services. The aspect weaver is indirectly also invoked based on execution of the same skeleton elements. Through the execution of advice it either modifies the service instances selected by the composer or it contributes additional constituent services. Both together determine the overall service serialization of the composition. This means that the proposed AOP solution constitutes an additional composition mechanism within the same execution engine. It ultimately complements and extends the composition development paradigm.

6.2 Thesis Validation

The first thesis claims that AOP concepts are a useful addition to service composition. A prerequisite for this thesis to be valid would be that cross-cutting concerns exist. There were many examples provided showing that this is particularly the case for telecommunication applications and business environments.

Furthermore, the first thesis also implies that the proposed AOP solution provides constructs with suitable semantics for modular implementation of cross-cutting concerns. This was discussed and verified in Chapter 5.1 based on typical use case examples covering a broad variety of important development tasks. Using the introduced AOP mechanism a more modular implementation of the additional concerns could be reached compared to a conventional implementation.

The effort for writing the aspect was shown to be reasonably small and the base application could always be kept free of direct modifications. This contributes to the verification of the second thesis claiming that Aspect Oriented Programming can be used efficiently in telecommunication service environments. Efficient development processes and tools refers to short time to market and high re-use of components. With respect to Aspect Oriented Programming this is reached by a higher degree of modularization while reducing complexity. The easy to achieve modular implementation of additional concerns was demonstrated in Chapter 5.1. It shows that the proposed AOP additions to the composition engine would allow to easily reach modular implementations of typical cross-cutting concerns. This partly verifies the claim of thesis two.

Run time characteristics with controlled short execution latencies are a specific and highly important requirement for telecommunication applications. In order to fully validate thesis two it needs to be shown that the use of aspect oriented programming still allows to keep the required execution characteristics. The Ericsson Composition Engine was particularly developed to show excellent performance in terms of composition latency. Because of the importance of these requirements, the composition performance when AOP is used became the most elaborate validation activity.

The validation in Chapter 5.2 was based on measurements of the execution time needed by example composite applications. Additionally, a mathematical model of the execution time contributing functions of the composition and weaving engine was developed. This allowed to distinguish specific contributions to execution time from basic composition, major actions in the weaving process and the advice execution.

The result of this performance evaluation first of all shows that weaving decisions can become the predominant contributor to execution time. Especially the repeated evaluation of all weaving instructions at each join-point causes a vast increase of composition execution time. Considering real uses cases with typical constituent services rather than only the composition process, the time spent on constituent service execution would still exceed combined composition and online weaving execution by a great extent. The details depend on the concrete use case, the performance of external protocol stacks and the time needed by the constituent service for processing. In typical cases a composition overhead of about 10% can be reached using AOP instead of about 2% with only the base composition process. If this is a too high latency cost for using AOP cannot be answered in general. It depends to a great extent on the application and its context.

One conclusion of this thesis is that dynamic AOP causes the composition execution process to be not lean any more. The validation in Chapter 5.2 allows to detect particular contributions of composition and weaving sub-actions while executing composite applications. It shows that the biggest contributer to composition latency is the repeated check of weaving conditions. The global character of weaving instructions causes that all weaving instructions are valid for each join-point with many unnecessary checks. The problem increases with the number of weaving instructions loaded into the engine. This means that the performance will drop significantly once AOP is seriously used in productive operation with many aspects applied.

The AOP system proposed by this dissertation is conceptually consistent and comprehensive in implementing the idea of dynamic online weaving. However, it could be shown, that this solution is not the most efficient with respect to usage of valuable processing resources. Therefore, this dissertation proposes variations to the original solution that would lead to improved execution efficiency of aspect weaving and execution. Some of these proposals reduce AOP features. One example is the removal of actions with low practical use, such as weaving within advice code. Additionally, weaving condition checks are moved from being executed at run time to design and deployment of applications and aspects. This is possible without limiting the definition of point-cuts in aspect design, and therefore without imposing limitations to the AOP concept. These changes show a substantial effect on improving the performance of the aspect weaving process.

Table 5.13 in Chapter 5.3 summarizes the improvements that could be reached by a particular optimization. All used in combination would improve the execution performance, and therefore the composition lead-time of an example composite application, by 73%.

This discussion leads directly to the validation of thesis 2 and the question if the run time requirements of the telecommunication domain can be met if aspects are used? A general answer is not possible. It depends on the context of the application, the amount of aspects that are applied and it depends in particular on the constituent services that are

used. The overall effort in terms of lead-time needed for composition execution including weaving and advice handling stays in the range of a few milliseconds. Although AOP imposes a much higher lead-time cost compared to pure service composition without aspect weaving, it appears to be still acceptable even for real-time telecommunication actions within end-to-end session set-up. The dominant contribution to overall execution time still comes from constituent services. This means, that the usage of aspects appears to have only a relatively small, and therefore acceptable, overall impact. With the proposed enhancements this statement is even more valid, because the dependency on the extent of AOP usage becomes significantly smaller. However, there might be situations where the small overall increase in lead time is not acceptable. If this is the case it is not possible to use dynamic online weaving.

Overall, also thesis two can be considered verified at least for many practical use cases. It is however highly recommended to analyze the detailed impact of aspect usage for a particular service composition environments and match it with its run time requirements. This dissertation has developed and quantified a theoretical model of the composition execution. This will be useful for assessing if the usage of aspect would be acceptable.

6.3 Research Results and Applicability

Prior to this dissertation, there were already a few proposals of how to do Aspect Oriented Programming for service composition environments. These proposals add aspect oriented programming to multi-purpose and mainstream composition technologies, such as BPEL. This dissertation focuses on the particular needs of telecommunication service environments. This has two consequences: First of all a composition technology is chosen as base, which has been designed particularly for the specific needs of telecommunication services. The second consequence is that composition performance is a particular interest.

This dissertation provides reasoning about the best choice of AOP variants and features in order to meet the unique needs of composite telecommunication applications. This includes a detailed discussion of semantics for the aspect programming model and needed abilities of the aspect weaving mechanism. In this respect, a highly important requirement was the operation in heterogeneous service environments. Furthermore, the related constraints based composition mechanism is not broken by the introduction of aspect weaving and execution. The resulting contribution is a combined service composition and aspect weaving engine. It provides tools for using aspects in the development and maintenance of telecommunication applications. A suitable join-point model and weaving language for dynamic online weaving was developed.

Furthermore, the role of aspects within the full application usage context in telecommunication sessions is shown. Especially the IMS/SIP based service usage patterns of end-to-end telecommunication sessions is a basic principle the composition and AOP solution need to support. Here telecommunication services differ considerably from other service technologies. The use of AOP in this context was demonstrated.

A unique aspect of the contribution is a solution for making run time-data available

to weaving and advice execution. The composite application's state and run time data is exposed in a way that is balancing protection of data, separation of execution sessions and broad data access. Next to the full composition session state, also internal run time data from composition execution is made available to weaving logic and advice sessions. All data is exposed in a uniform way to advice and weaving without the need of introducing new constructs in the composition language.

The main contribution of this work is however a thorough performance examination of the proposed AOP enabled composition mechanism. This methods used in the evaluation allow identifying the extent to which each major component of the implementation contributes to the composite service execution time. It furthermore discusses variants in the implementation and quantifies the expected performance impact. This provides valuable insights allowing future AOP implementations to be optimized with the right balance of features and performance.

These results can be directly used as foundation for designing frameworks and tools for service application development in the telecommunication domain. In this respect, the extensive performance validation of the proposed solution and major variants of it will provide valuable guidance for meeting domain specific requirements with the right balance of features versus performance. The proof-of-concept implementation is already complete in the sense that it contains a baseline of features for productive use of aspect oriented programming. Next to the execution engine for combined service composition and aspect weaving, a basic, but fully functional set of tools for development and management of aspects is demonstrated.

While the originally chosen application domain is telecommunication, most properties of the AOP solution do not have a specific dependency to it. They are rather universally applicable to whatever domain has similar requirements. With its unique characteristics the Ericsson Composition Engine already meets several goals defined as core enablers in pervasive computing [18] and for the Internet of Things [19]: For example, the flexibility of constraints based service selection allows managing contingencies by automatically adapting the choice of constituent services to those momentarily available in the environment. It similarity also allows dynamic adaptation of the composite application to various contexts as described in Chapter 2.2.10. Furthermore, the ability to natively compose services from various technological backgrounds within a single composite application means that a great variety of existing components can be utilized. This directly addresses the expected technological diversity in IoT and pervasive computing.

An Aspect Oriented Programming solution, as introduced by this dissertation, contributes yet another degree of flexibility and adaptability towards the goals of pervasive computing. In particular, addition of advice through run time weaving allows a high degree of context dependent changes in the behavior of composite applications. In this respect aspect oriented programming introduces the technological foundation for applying dynamic changes. The inference logic for deciding about needed modifications, and when they need to be applied, resides on two levels: The weaving mechanism establishes an overlay to the composition semantics, which resides outside the composite application. It can therefore be controlled separately from the composition design. This leads to the second level of con-

textual adaptation: An aspect management system can modify the assignment of aspects by means of rules for changing weaving instructions. Aspects can temporarily and selectively be added to or removed from composite applications or single composition sessions. These features of the AOP solution were discussed in Chapter 3.5. They also provide a technical foundation for more elaborate contextual modification logic, for example based on machine reasoning within semantic world models and ontologies.

These characteristics of the investigated solution make this dissertation and tits results highly versatile and relevant for pervasive computing and contextual composition. These emerging domains are directly connected to technical areas, such as IoT and cyber-physical systems, which are expected to dramatically enrich the service landscape and change the way users interact with the surrounding technical environment. Service composition, and in particular flexible contextual adaptability built into it, will be a major technical enabler of complex environment in which users, things and services interact with each other. This is expected to have a disruptive impact on our society and human interaction.

In this context the first statement in Chapter 1 appears to be as relevant for the future as it was ever: "Communication is a basic human need". This dissertation contributes directly to this future. The requirements for the developed solutions were originally derived from telecommunication use cases. They are to a great extent the same requirements that govern these emerging domains, which determine the future of human communication. Thus, this dissertation provides direct advice for essential architectural and conceptual decisions, when designing the service management and execution systems of the future.

6.4 Future Work

This dissertation provides a couple indications for future research and development work. The most important one is directly related to the proposed performance improvements. There are strong indications that a hybrid weaving engine supporting online and offline weaving combined, could lead to a well balanced solution from performance and functional point of view. It would provide all flexibility of dynamic online weaving if needed, but run time performance degradation could be minimized for those concerns, which do not need dynamic behavior. They can be applied offline and prior to any application execution. This should be further investigated. Is it, for example, possible to develop aspects, which can directly be used in both weaving variants? What would be performance in typical use cases.

The validation of thesis 1 regarding the usefulness of the concept is based on experiments with typical example use cases as described in Chapter 5.1. It focuses on a selection of essential programming tasks when implementing composite applications. It demonstrates that modification of these essential parts of a composite application are possible by utilizing the presented AOP concepts. The use cases were chosen by the author without an independent evaluation. It would be beneficial to extend this investigation in future research and perform an independent selection of use cases and their broad assessment by a bigger set of developers. This would allow an independent assessment of the usefulness

and completeness of the proposed concepts.

Highly related to the choice of weaving method is the general life-cycle management for advice and the related concerns. This dissertation has only proposed and implemented quite rudimentary management features. Weaving instructions can be loaded and then individually enabled or disabled. It would be beneficial to investigate the possibilities of a more elaborate aspect management as briefly described in Chapter 3.5. A solution could be investigated that dynamically enables or disables advice based on reflection about the entire service execution environment and its diverse context. Rules might be introduced in order to implement a policy by means of a flexible weaving strategy. Context information would allow to identify various contextual situations and apply the appropriate policy through dynamic aspects.

A great variety of data source could be used in order to establish context and constitute weaving conditions. For example, a data warehouse might contain run time performance characteristics of all services or policies that need to be followed. Based on this kind of information the weaving engine could dynamically decide, which aspects would be needed and load or remove the respective weaving instructions. A function, which automatically manages those concerns addressed within the application is another example of an aspect management function.

Furthermore, a dynamic AOP environment is characterized by aspect interaction problems, such as a sensible order of advice execution. Several aspects, applied at the same join-point, are likely to break each other, if they also modify the same base application elements. In the proof-of-concept implementation provided by this thesis, the application of aspects depends on the load order of the weaving instructions. There is no further management entity for detecting and solving inter advice dependencies. These aspect interaction problems still needs to be solved manually. The resulting problems should be studied in greater detail, potentially resulting in suitable concepts for managing advice interaction.

This work has briefly outlined a concern management mechanism in Chapter 3.5. It does bookkeeping of all concerns, which are addressed by an application or by already added aspects. Applying further aspects for already addressed concerns would be blocked. This concern management idea could be further developed into a full aspect management solution.

The Ericsson Composition Engine is a particular solution for service composition specifically made for telecommunication application composition. There are several other composition solution available based, for example, on BPEL or BPMN. These are more mainstream and general purpose and AOP solutions were also developed for them. It would be interesting to investigate their performance in detail when being used for telecommunication applications. For a full functional equivalent, this might however require that semantics for telecommunication services, as for example defined in SIP, need to be added to the composition execution engine and potentially also the composition language.

Ericsson has proposed a new composition language called SCALE [88]. it specifically addresses service composition, where a single application with many parallel activity threads can be developed and efficiently executed. An AOP solution for this language bears particular challenges from the parallel and asynchronous execution nature of constituent services.

Aspects in this kind of environment might introduce, for example, new race conditions and potentially a new dimension of interaction problems.

Another interesting question would be, to which extend fragile point-cuts might still be a problem once the abstraction of the application development process rises. The idea is that higher abstraction of a purely data driven composition model can provide robust point-cuts. The fragility is still present, but it might exist only in the higher level of abstraction, where it can be managed more easily. The advantaged for aspects, which are weaved based on abstract models of the application, can already be assumed from the discussion of abstract service selection constraints. For certain use cases, when advice is weaved in based on these constraints only, aspect appears to be much more robust and adaptive as long as the constraint model does not change. This could be explored in greater detail.

These examples show that there are a lot of research challenges in the area of Aspect Oriented Programing that first of all will lead to a better understanding the application and behavior of aspects. Furthermore it will have a direct impact on making good use of AOP in real-life production environments.

Appendix A

Acronyms

3GPP	3rd Generation Partnership Project
AJAX	Asynchronous JavaScript and XML
AOP	Aspect Oriented Programming
AOSD	Aspect Oriented Software Development
API	Application Programming Interface
AuC	Authentication Center
AS	Application Server
BG	Border Gateway
BICC	Bearer Independent Call Control
BPEL	Business Process Execution Language
BPMN	Business Process Modeling Notation
BSS	Business Support Systems
CAMEL	Customized Applications for Mobile networks Enhanced Logic
CAP	CAMEL Application Part
CAPEX	Capital Expenditure
CBSD	Component Based Software Development
CE	Composition Engine
CEA	Composition Execution Agent
CRM	Customer Relationship Management
CS	Circuit Switched
DNS	Domain Name Service
EA	Execution Agent (same as CEA)
ECA	Event-Condition-Action
ECE	Ericsson Composition Engine
EJB	Enterprise Java Beans
ESB	Enterprise Service Bus
ETSI	European Telecommunication Standards Institute
GGSN	Gateway GPRS Support Node
GMSC	Gateway Mobile services Switching Center
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communication
HLR	Home Location Register
HSS	Home Subscriber Server
HTTP	Hypertext Transfer Protocol
ICT	Information and Communication Technologies
IETF	Internet Engineering Task Force
iFC	initial Filter Criteria
IM-SSF	IMS Service Switching Function
IMS	IP Multimedia Sub-system
IMSI	International Mobile Subscriber Identity
IN	Intelligent Network
INAP	Intelligent Network Application Part

IoT	Internet of Things
IP	Internet Protocol
IPv4	IP Version 4
IPv6	IP Version 6
ISC	IMS Service Control
ISDN	Integrated Services Digital Network
ISUP	ISDN User Part
IT	Information Technology
IVR	Interactive Voice Recognition
JEE	Java Enterprise Edition
JSR	Java Standardization Request
KPI	Key Performance Indicator
LDAP	Lightweight Directory Access Protocol
LTE	3GPP Long Term Evolution
MGCF	Media Gateway Control Function
MGW	Media Gateway
MMTEL	Multi-Media Telephony Service
MAP	Mobile Application Part
MSC	Mobile services Switching Center
MSISDN	Mobile Subscriber International ISDN Number
NGN	Next Generation Network
OPEX	Operating Expenditure
OSA	Open Services Access
OSS	Operational Support Systems
PHP	PHP Hypertext Preprocessor
PLMN	Public Land Mobile Network
PS	Packet Switched
PSTN	Public Switched Telephone Networks
QoS	Quality of Service
REST	Representational State Transfer
RMI	Remote Message Invocation
RPC	Remote Procedure Call
SCE	Service Creation Environment
SCIM	Service Capabilities Interaction Manager
SCTP	Stream Control Transmission Protocol
SCP	Service Control Point
SGSN	Serving GPRS Support Node
SGW	Signaling Gateway
SIP	Session Initiation Protocol
SLA	Service Level Agreement
SMS	Short Message Service
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SS7	Signaling System No. 7
SSF	Service Switching Function
SSM	Shared State Manager
TCP	Transmission Control Protocol
TDM	Time Division Multiplex
TISPAN	ETSI Telecoms and Internet converged Services and Protocols for Advanced Networks
UA	User Agent
UAC	User Agent Client
UAS	User Agent Server
UDDI	Universal Description Discovery and Integration of Web Services
UDP	User Datagram Protocol
UMTS	Universal Mobile Telecommunications System
UTRAN	UMTS Terrestrial Radio Access Network
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VLR	Visitor Location Register
VoIP	Voice over IP
WS	Web Service
WS-BPEL	Web Services Business Process Execution Language
WSDL	Web Services Description Language
XML	Extensible Markup Language

Bibliography

[1] J. Meurling and R. Jeans, *The Ericsson Chronicle, 125 Years in Telecommunication*. Informationsförlaget, 2000.

[2] J. Davies, A. Duke, S. Stini Clarke, N. Mehandjiev, and G. lvaro Rey, "Chasing the long tail: Growth through personalized telecoms services," in *Case Studies in Service Innovation*, ser. Service Science: Research and Innovations in the Service Economy, L. A. Macaulay, I. Miles, J. Wilby, Y. L. Tan, L. Zhao, and B. Theodoulidis, Eds. Springer New York, 2012, pp. 133–136.

[3] C. Szyperski, *Component Software: Beyond Object-oriented Programming*. ACM Press/Addison-Wesley Publishing Co., 1998.

[4] M. P. Papazoglou, *Web Services: Principles and Technology*, ser. Pearson education. Pearson Prentice Hall, 2008.

[5] M. P. Papazoglou, *Web Services and SOA: Principles and Technology*. Pearson Education, Limited, 2012.

[6] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, "BPEL4WS, Business Process Execution Language for Web Services Version 1.1," IBM, Standard, 2003. [Online]. Available: http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf

[7] OMG, "Business Process Model and Notation (BPMN), Version 2.0," Object Management Group, Standard, Jan. 2011. [Online]. Available: http://www.omg.org/spec/BPMN/2.0

[8] T. Dinsing, G. Eriksson, I. Fikouras, K. Gronowski, R. Levenshteyn, P. Pettersson, and P. Wiss, "Service composition in ims using java ee sip servlet containers," vol. 84, no. 3, pp. 92–96. [Online]. Available: http://www.ericsson.com/ericsson/corpinfo/publications/review/2007_03/files/4_ServiceComposition.pdf

[9] J. Niemöller, I. Fikouras, F. de Rooij, L. Klostermann, U. Stringer, and U. Olsson, "Ericsson composition engine - next-generation in," vol. 86, no. 2, pp. 22–27, September 2009. [Online]. Available: http://www.ericsson.com/res/thecompany/docs/publications/ericsson_review/2009/issue2/ngin.pdf

[10] J. Niemöller, E. Freiter, K. Vandikas, R. Quinet, R. Levenshteyn, and I. Fikouras, "Multi-technology service composition for the telecommunication domain - concepts and experiences," in *Proceedings of the 2010 Fourth International Conference on Next Generation Mobile Applications, Services and Technologies*, ser. NGMAST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 34–41.

[11] K. Vandikas, E. Freiter, R. Levenshteyn, R. Quinet, J. Niemöller, and I. Fikouras, "Blending the telecommunication domain with web 2.0 services," in *Proceedings of the 2010 14th International Conference on Intelligence in Next Generation Networks (ICIN)*, ser. ICIN '10. IEEE Computer Society, 2010, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1109/NGMAST.2010.19

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the European Confefence on Object-Oriented Programming (ECOOP)*, ser. Lecture Notes in Computer Science, M. Aksit and S. Matsuoka, Eds., vol. 1241. Springer-Verlag, 1997, pp. 220–242.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *Proceedings of the 15th European Conference on Object-Oriented Programming*, ser. ECOOP '01. London, UK, UK: Springer-Verlag, 2001, pp. 327–353.

[14] C. V. Lopes and G. Kiczales, "Improving design and source code modularity using AspectJ (tutorial session)," in *Proceedings of the 22Nd International Conference on Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 825–.

[15] J. Armstrong, "A history of erlang," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL III. New York, NY, USA: ACM, 2007, pp. 6–1–6–26. [Online]. Available: http://doi.acm.org/10.1145/1238844.1238850

[16] R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in ERLANG (2Nd Ed.)*, J. Armstrong, Ed. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996.

[17] R. Noldus, *CAMEL: Intelligent Networks for the GSM, GPRS and UMTS Network*. John Wiley & Sons, 2006.

[18] J. Bronsted, K. Hansen, and M. Ingstrup, "Service composition issues in pervasive computing," *Pervasive Computing, IEEE*, vol. 9, no. 1, pp. 62–70, Jan 2010.

[19] T. Teixeira, S. Hachem, V. Issarny, and N. Georgantas, "Service oriented middleware for the internet of things: A perspective," in *Towards a Service-Based Internet*, ser. Lecture Notes in Computer Science, 2011, vol. 6994, pp. 220–229.

[20] A. G. Bell, "Improvement in telegraphy," USA Patent US174 465, Mar. 07, 1876.

[21] 3GPP, "Customized Applications for Mobile network Enhanced Logic (CAMEL); Service description; Stage 1," 3rd Generation Partnership Project (3GPP), TS 22.078, Dec. 2005. [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/22078.htm

[22] 3GPP, "Customized Applications for Mobile network Enhanced Logic (CAMEL) Phase X; Stage 2," 3rd Generation Partnership Project (3GPP), TS 23.078, Sep. 2007. [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/23078.htm

[23] 3GPP, "Customised Applications for Mobile network Enhanced Logic (CAMEL) Phase 4; Stage 2; IM CN Interworking," 3rd Generation Partnership Project (3GPP), TS 23.278, Mar. 2006. [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/23278.htm

[24] R. Noldus, U. Olsson, C. Mulligan, I. Fikouras, A. Ryde, and M. Stille, *IMS Application Developer's Handbook: Creating and Deploying Innovative IMS Applications*, 1st ed. Academic Press, 2011.

[25] 3GPP, "Customized Applications for Mobile network Enhanced Logic (CAMEL) Phase X; CAMEL Application Part (CAP) specification," 3rd Generation Partnership Project (3GPP), TS 29.078, Sep. 2007. [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/29078.htm

[26] 3GPP, "Customized Applications for Mobile network Enhanced Logic (CAMEL); CAMEL Application Part (CAP) specification for IP Multimedia Subsystems (IMS)," 3rd Generation Partnership Project (3GPP), TS 29.278, Dec. 2005. [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/29278.htm

[27] G. Camarillo and M.-A. Garcia-Martin, *The 3G IP Multimedia Subsystem (IMS): Merging the Internet and the Cellular Worlds*, 2nd ed. John Wiley & Sons, 2006.

[28] 3GPP, "Network architecture," 3rd Generation Partnership Project (3GPP), TS 23.002, Sep. 2008. [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/23002.htm

[29] 3GPP, "IP Multimedia Subsystem (IMS); Stage 2," 3rd Generation Partnership Project (3GPP), TS 23.228, Sep. 2008. [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/23228.htm

[30] 3GPP, "TISPAN; IP Multimedia Subsystem (IMS); Stage 2 description [3GPP TS 23.228 v7.2.0, modified]," 3rd Generation Partnership Project (3GPP), TS 23.406, Sep. 2008. [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/23406.htm

[31] 3GPP, "TISPAN; IP Multimedia Subsystem (IMS); Functional architecture," 3rd Generation Partnership Project (3GPP), TS 23.417, Dec. 2007. [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/23417.htm

[32] H. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg, "SIP: Session Initiation Protocol," Internet Engineering Task Force (IETF), RFC 2543, Mar. 1999. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2543.txt

[33] G. Camarillo, *SIP Demystified*. McGraw-Hill Professional, 2001.

[34] Project Sailfin, "Sailfin 2.0," 2013. [Online]. Available: https://sailfin.java.net/

[35] Oracle, "Glassfish Java EE7 Application Server," 2013. [Online]. Available: https://glassfish.java.net/

[36] 3GPP, "IP Multimedia (IM) session handling; IM call model; Stage 2; Release 7," 3rd Generation Partnership Project (3GPP), TS 23.218, Dec. 2005. [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/23218.htm

[37] 3GPP, "Open Service Access (OSA) Application Programming Interface (API); Part 1: Overview," 3rd Generation Partnership Project (3GPP), TS 29.198-01, Mar. 2007. [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/29198-01.htm

[38] R. Mordani, "Java Servlet Specification," Sun Microsystems, Inc., Java Standardization Request JSR-315, Dec. 2009. [Online]. Available: https://jcp.org/en/jsr/all

[39] A. Kristensen, "SIP Servlet API," Sun Microsystems, Inc., Java Standardization Request 116, Feb. 2003. [Online]. Available: https://jcp.org/en/jsr/all

[40] M. Kulkarni and Y. Cosmadopoulos, "SIP Servlet API v1.1," Sun Microsystems, Inc., Java Standardization Request JSR-289, Aug. 2008. [Online]. Available: https://jcp.org/en/jsr/all

[41] I. Fikouras, "An approach to service composition for internet and mobile services based on knowledge-based model-driven variant configuration," PhD Thesis, University of Bremen, Germany, 2008.

[42] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, May 2002.

[43] A. T. Holdener, *AJAX - The Definite Guide*, 1st ed. OReilly Media, Inc., Jan. 2008.

[44] L. Clement, A. Hately, C. von Riegen, and T. Rogers, "UDDI Version 3 Specifications," Advancing open standards for the information society (OASIS), standard, Feb. 2005. [Online]. Available: https://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv3

[45] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana, "Web services description language (wsdl) version 2.0 part 1: Core language," World Wide Web Consortium (W3C), standard, Jun. 2007. [Online]. Available: http://www.w3.org/TR/wsdl/

[46] N. Mitra and Y. Lafon, "Simple Object Access Protocol Version 1.2," World Wide Web Consortium (W3C), standard, Apr. 2007. [Online]. Available: http://www.w3.org/TR/soap/

[47] I. Fikouras and E. Freiter, "Service discovery and orchestration for distributed service repositories," in *Service-Oriented Computing - ICSOC 2003*, ser. Lecture Notes in Computer Science, M. Orlowska, S. Weerawarana, M. P. Papazoglou, and J. Yang, Eds. Springer Berlin Heidelberg, 2003, vol. 2910, pp. 59–74.

[48] L. Ferreira Pires, N. Maatjes, M. van Sinderen, and P. Dockhorn Costa, "Model-driven approach to the implementation of context-aware applications using rule engines," in *Constructing Ambient Intelligence*, ser. Communications in Computer and Information Science, M. Mhlhuser, A. Ferscha, and E. Aitenbichler, Eds., 2008, vol. 11, pp. 104–112.

[49] P. Dockhorn Costa, J. P. Andrade Almeida, L. Ferreira Pires, and M. van Sinderen, "Evaluation of a rule-based approach for context-aware services," in *2008 IEEE Global Telecommunications Conferences, GlobeCom 2008*. Red Hook, NY, USA: IEEE, Nov 2008, pp. 1657–1661. [Online]. Available: http://doc.utwente.nl/62633/

[50] M. Vukovi, "Context aware service composition," University of Cambridge Computer Laboratory, Tech. Rep., Oct 2007. [Online]. Available: http://www.cl.cam.ac.uk/techreports/

[51] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, pp. 1053–1058, 1972.

[52] K. Lieberherr, D. Lorenz, and M. Mezini, "Programming with aspectual components," College of Computer Science, Northeastern University, Boston, MA, Tech. Rep. NU-CCS-99-01, Mar. 1999.

[53] K. J. Lieberherr, "Controlling the complexity of software designs," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. IEEE Computer Society, May 2004, pp. 2–11.

[54] K. Lieberherr, D. H. Lorenz, and J. Ovlinger, "Aspectual collaborations: Combining modules and aspects," *The Computer Journal*, vol. 46, p. 2003, 2003.

[55] H. Masuhara, G. Kiczales, and C. Dutchyn, "A compilation and optimization model for aspect-oriented programs," in *Proceedings of the 12th International Conference on Compiler Construction*, ser. CC'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 46–60.

[56] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "Getting started with AapectJ," *Communications of the ACM*, vol. 44, no. 10, pp. 59–65, Oct. 2001.

[57] A. Colyer and A. Clement, "Aspect-oriented programming with AspectJ," *IBM Systems Journal*, vol. 44, no. 2, pp. 301–308, Jan. 2005.

[58] R. Laddad, *AspectJ in Action: Enterprise AOP with Spring Applications*, 2nd ed. Greenwich, CT, USA: Manning Publications Co., 2009.

[59] E. Hilsdale, "Aspect-oriented programming with AspectJ," in *TOOLS (39)*. IEEE Computer Society, 2001, p. 368.

[60] The Eclipse Foundation, "AspectJ Project," Feb. 2014. [Online]. Available: http://www.eclipse.org/aspectj/

[61] The AspectJ Team, "The AspectJ Programming Guide," Feb. 2014. [Online]. Available: http://eclipse.org/aspectj/doc/released/progguide/index.html

[62] Y. Sato, S. Chiba, and M. Tatsubori, "A selective, just-in-time aspect weaver," in *Generative Programming and Component Engineering*, ser. Lecture Notes in Computer Science, F. Pfenning and Y. Smaragdakis, Eds. Springer Berlin Heidelberg, 2003, vol. 2830, pp. 189–208.

[63] JBoss Community. (2014, Feb.) JBoss AOP - Framework for Organizing Cross Cutting Concerns. [Online]. Available: http://www.jboss.org/overview/

[64] D. Schweisguth. (2004, Jul.) Second-generation aspect-oriented programming. [Online]. Available: http://www.javaworld.com/article/2072818/core-java/second-generation-aspect-oriented-programming.html

[65] M. Fleury and F. Reverbel, "The jboss extensible server," in *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, ser. Middleware '03. Springer-Verlag New York, Inc., 2003, pp. 344–373.

[66] J. Baker and W. Hsieh, "Runtime aspect weaving through metaprogramming," in *Proceedings of the 1st International Conference on Aspect-oriented Software Development*, ser. AOSD '02. ACM, 2002, pp. 86–95.

[67] K. Böllert, "On weaving aspects," in *Proceedings of the Workshop on Object-Oriented Technology*. London, UK, UK: Springer-Verlag, 1999, pp. 301–302.

[68] A. Popovici, T. Gross, and G. Alonso, "Dynamic weaving for aspect-oriented programming," in *In Proceedings of the 1st International Conference on Aspect-Oriented Software Development*. ACM Press, 2002, pp. 141–147.

[69] A. Colyer and A. Clement, "Large-scale AOSD for middleware," in *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, ser. AOSD '04. ACM, 2004, pp. 56–65.

[70] S. G. Thompson and B. Odgers, "Aspect-oriented process engineering," in *Proceedings of the Workshop on Object-Oriented Technology*. Springer-Verlag, 1999, pp. 295–.

[71] B. Bachmendo and R. Unland, "Aspect-based workflow evolution," in *In Proc. of the Workshop on Aspect-Oriented Programming and Separation of Concerns*, 2001.

[72] D. Suvée, W. Vanderperren, and V. Jonckers, "Jasco: An aspect-oriented approach tailored for component based software development," in *Proceedings of the 2Nd International Conference on Aspect-oriented Software Development*, ser. AOSD '03. ACM, 2003, pp. 21–29.

[73] G. Hamilton, "JavaBeans Specification," Sun Microsystems, Specification, Aug. 1997.

[74] M. A. Cibrán, B. Verheecke, W. Vanderperren, D. Suvée, and V. Jonckers, "Aspect-oriented programming for dynamic web service selection, integration and management," *World Wide Web*, vol. 10, no. 3, pp. 211–242, sep 2007.

[75] A. Charfi and M. Mezini, "Aspect-oriented web service composition with AO4BPEL," in *Web Services*, ser. Lecture Notes in Computer Science, L.-J. Zhang and M. Jeckle, Eds. Springer Berlin Heidelberg, 2004, vol. 3250, pp. 168–182.

[76] A. Charfi and M. Mezini, "Aspect-oriented workflow languages," in *In proceeding of: On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, ser. OTM Confederated International Conferences, 2006, pp. 183–200.

[77] A. Charfi and M. Mezini, "AO4BPEL: An aspect-oriented extension to BPEL," *World Wide Web*, vol. 10, no. 3, pp. 309–344, Sep. 2007.

[78] A. Charfi, B. Schmeling, and M. Mezini, "Transactional BPEL processes with AO4BPEL aspects," in *Fifth European Conference on Web Services, 2007. ECOWS '07.* IEEE Computer Society, Nov. 2007, pp. 149–158.

[79] D. Karastoyanova and F. Leymann, "BPEL'n'Aspects: Adapting service orchestration logic," in *IEEE International Conference on Web Services, 2009. ICWS 2009.* IEEE, July 2009, pp. 222–229.

[80] M. Sonntag and D. Karastoyanova, "Compensation of adapted service orchestration logic in BPEL'n'Aspects," in *Proceedings of the 9th International Conference on Business Process Management*, ser. BPM'11. Springer-Verlag Berlin, Heidelberg, 2011, pp. 413–428.

[81] M. Sonntag and D. Karastoyanova, "BPEL'n'Aspects & compensation: Adapted service orchestration logic and its compensation using aspects," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, vol. 6470, pp. 724–725.

[82] A. Osterwalder and Y. Pigneur, *Business Model Generation: A Handbook for Visionaries, Game Changers, and Challengers*, ser. Business Model Generation: A Handbook for Visionaries, Game Changers, and Challengers. John Wiley & Sons, 2010.

[83] A. Kellens, K. Gybels, J. Brichau, and K. Mens, "A model-driven pointcut language for more robust pointcuts," in *Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT! 2006) collocated with AOSD 2006*, 2006, p. 7.

[84] A. Kellens, K. Mens, J. Brichau, and K. Gybels, "Managing the evolution of aspect-oriented software with model-based pointcuts," in *In Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Spring-Verlag, 2006, pp. 501–525.

[85] Red Hat inc. DRools project homepage. [Online]. Available: http://www.drools.org/

[86] J. Niemöller, R. Levenshteyn, E. Freiter, and R. Quinet, "Application server and method for managing a service," U.S. Patent Application No. 13/511,250, May 22, 2012. [Online]. Available: http://patentscope.wipo.int/search/en/WO2011064001

[87] J. Niemöller, R. Quinet, R. Levenshteyn, and I. Fikouras, "Cost control in service composition environments," in *Proceedings of the 2008 The Second International Conference on Next Generation Mobile Applications, Services, and Technologies*, ser. NGMAST '08. IEEE Computer Society, 2008, pp. 25–32.

[88] J. Niemöller and K. Vandikas, "SCALE - a language for dynamic composition of heterogeneous services," Ericsson AB, Technical Report, 2010. [Online]. Available: http://www.ericsson.com/res/thecompany/docs/journal_conference_papers/service_layer/101215_scale.pdf

List of Figures

List of Tables

Curriculum Vitae and Summary

Jörg Niemöller was born in Dortmund, Germany in 1970. He obtained his Diploma in electrical engineering from the University of Dortmund in 1998. His thesis "Spectral Broadening of External Cavity Lasers" was executed at Hewlett Packard in Böblingen, Germany. In 1998 Jörg joined research and development of Ericsson in Herzogenrath, Germany. He developed solutions for the introduction of 3G/UMTS and became expert for charging solutions in mobile core networks. Jörg joined Ericsson Research in 2006. He contributed service level agreement awareness and billing strategies for composite services and the integration of an Aspect Oriented Programming framework into the Ericsson Composition Engine. In 2011 Jörg moved to Stockholm, Sweden taking a new position within Ericsson Research. His focus changed to the Internet of things combined with semantic models, machine reasoning and machine learning. In 2013 Jörg took a leading role in the development of a new product line for customer experience management and big data analytics. His particular interest became predictive analytics of customer sentiment and machine reasoning with semantic models for automated bridging between technical infrastructure and business needs. Jörg represents Ericsson at TM Forum where he is leading collaborative standardization projects. He holds multiple patents in service composition, charging & billing, the Internet of things, cloud computing and data analytics.

This PhD dissertation investigates how to overcome the negative effects of cross cutting concerns in the development of composite service applications. It proposes a combination of dynamic aspect oriented programming with a rules driven service composition mechanism. This combination enables very flexible utilization of aspects based on run-time data. The thesis contributes a join-point model and it integrates techniques for weaving and advice definition into the underlying composition language and execution engine. A particular focus is telecommunication applications with their unique model for utilizing heterogeneous constituent services and severe real-time requirements. In addition to its primary focus on modular implementation and dynamic deployment of concerns in telecommunication services, the dissertation discusses AOP as an enabler for automated management and customization of applications. The verification of the proposed solution contributes a detailed assessment of run-time performance. This includes a theoretical model of the AOP implementation for a quantitative prediction of performance expected from alternative AOP solutions. The combination of AOP and service composition proposed by this dissertation provides features, which directly address challenges in pervasive computing and the Internet of things. Thus, this dissertation advances beyond telecommunication with results applicable to various highly relevant technical domains.

ISBN: 978 90 5668 468 6